МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Изучаем программирование на Python

2-е издание

Все, что нужно знать о Python

Строй списки, словари, кортежи и множества

Не мучайся – используй DB-API

Объекты? Декораторы? Генераторы? Они все здесь

Создавай веб-приложения с Flask

Записывай код при помощи модулей

Пол Бэрри

МИРОВОЙ
КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Head First

Python

Paul Barry

O'Reilly

МИРОВОЙ
КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Изучаем программирование на Python

Пол Бэрри

Москва
2017

УДК 004.43
ББК 32.973.26-018.1
Б97

Authorized Russian translation of the English edition of Head First Python,
2nd Edition (ISBN 9781491919538) © 2016 Paul Barry. This translation is published and sold
by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Бэрри, Пол.

Б97 Изучаем программирование на Python / Пол Бэрри ; [пер. с англ. М.А. Райтман]. — Москва : Издательство «Э», 2017. — 624 с. : ил. —(Мировой компьютерный бестселлер).

ISBN 978-5-699-98595-1

Надоело продираться через дебри малопонятных самоучителей по программированию? С этой книгой вы без труда усвоите азы Python и научитесь работать со структурами и функциями. В ходе обучения вы создадите свое собственное веб-приложение и узнаете, как управлять базами данных, обрабатывать исключения, пользоваться контекстными менеджерами, декораторами и генераторами. Все это и многое другое — во втором издании «Изучаем программирование на Python».

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-699-98595-1

© Райтман М.А., перевод на русский язык, 2017
© Оформление. ООО «Издательство «Э», 2017

Я посвящаю эту книгу всем бескорыстным членам сообщества Python, которые помогают этому языку идти в ногу со временем.

И всем тем, кто сделал изучение Python и связанных с ним технологий настолько сложным, что нужна *подобная* книга, чтобы справиться с ними.

Пол Бэрри: «Изучаем программирование на Python», 2-е издание

На прогулке Пол остановился, чтобы обсудить правильное произношение слова «tuple» со своей терпеливой женой.

Обычная реакция Дейдры ☺

Пол Бэрри живет и работает в *Карлоу* (*Ирландия*), маленьком городке с населением около 35 тысяч человек в 80 км на юго-запад от Дублина.

Пол имеет степень *бакалавра наук в области информационных систем* и степень *магистра в области вычислений*. Он также закончил аспирантуру и получил свидетельство на право *преподавания и обучения*.

Пол работает в Технологическом институте Карлоу с 1995 и читает лекции с 1997 года. Прежде чем начать преподавательскую деятельность, Пол десять лет посвятил ИТ-индустрии, работал в Ирландии и Канаде, большая часть его работы была связана с медицинскими учреждениями. Пол женат на Дейдре, у них трое детей (двое сейчас учатся в колледже).

Язык программирования Python (и связанные с ним технологии) составляют основу послевузовских курсов Пола с 2007 учебного года.

Пол является автором (или соавтором) еще четырех книг: двух о Python и двух о Perl. В прошлом он подготовил довольно много статей для *Linux Journal Magazine*, в котором является пишущим редактором.

Пол вырос в Белфасте, Северная Ирландия, и это во многом объясняет некоторые его взгляды и забавный акцент (впрочем, если вы тоже «с севера», тогда взгляды Пола и его акцент покажутся вам вполне нормальными).

Вы можете найти Пола в *Твиттере (@barrypj)*. У него есть также своя домашняя страничка *http://paulbarry.itcarlow.ie*.

Оглавление (краткое)

1	Основы. *Начнем поскорее*	*37*
2	Списки. *Работа с упорядоченными данными*	*83*
3	Структурированные данные. *Работа со структурированными данными*	*131*
4	Повторное использование. *Функции и модули*	*181*
5	Построение веб-приложения. *Возвращение в реальный мир*	*231*
6	Хранение и обработка данных. *Где хранить данные*	*279*
7	Использование базы данных. *Используем DB-API в Python*	*317*
8	Немного о классах. *Абстракция поведения и состояния*	*345*
9	Протокол управления контекстом. *Подключение к инструкции with*	*371*
10	Декораторы функций. *Обертывание функций*	*399*
11	Обработка исключений. *Что делать, когда что-то идет не так*	*449*
11¾	Немного о многопоточности. *Обработка ожидания*	*497*
12	Продвинутые итерации. *Безумные циклы*	513
A	Установка. *Установка Python*	*557*
B	Pythonanywhere. *Развертывание веб-приложения*	*565*
C	Топ-10 тем, которые мы не рассмотрели. *Всегда есть чему поучиться*	*575*
D	Топ-10 проектов, которые мы не рассмотрели. *Еще больше инструментов, библиотек и модулей*	*587*
E	Присоединяйтесь. *Сообщество Python*	*599*

Содержание (конкретное)

Введение

Ваш мозг и Python. Вы пытаетесь чему-то *научиться*, а мозг делает вам одолжение и *сопротивляется* изо всех сил. Он думает: «Лучше оставить место для запоминания действительно важных вещей. Вдруг нам встретится голодный тигр или захочется покататься голышом на сноуборде. Я должен помнить об опасности». Как же нам *обмануть* ваш мозг, чтобы он считал программирование на Python важным для выживания?

Для кого эта книга?	26
Мы знаем, о чем вы подумали	27
Мы знаем, о чем подумал ваш мозг	27
Метапознание: размышления о мышлении	29
Вот что мы сделали	30
Прочти меня	32
Команда технических редакторов	34
Признательности и благодарности	35

Основы

1 Начнем поскорее

Начнем программировать на Python как можно скорее.

В этой главе мы ознакомимся с основами программирования на Python и сделаем это в характерном для нас стиле: с места в карьер. Через несколько страниц вы запустите свою первую программу. К концу главы вы сможете не только запускать типичные программы, но также понимать их код (и это еще не все!). Попутно вы познакомитесь с некоторыми особенностями языка **Python**. Итак, не будем больше тратить время. Переверните страницу — и вперед!

Назначение окон IDLE	40
Выполнение кода, одна инструкция за раз	44
Функции + модули = стандартная библиотека	45
Встроенные структуры данных	49
Вызов метода возвращает результат	50
Принятие решения о запуске блока кода	51
Какие варианты может иметь «if»?	53
Блоки кода могут содержать встроенные блоки	54
Возвращение в командную оболочку Python	58
Экспериментируем в оболочке	59
Перебор последовательности объектов	60
Повторяем определенное количество раз	61
Применим решение задачи № 1 к нашему коду	62
Устраиваем паузу выполнения	64
Генерация случайных чисел на Python	66
Создание серьезного бизнес-приложения	74
Отступы вас бесят?	76
Попросим интерпретатор помочь с функцией	77
Эксперименты с диапазонами	78
Код из главы 1	82

Списки

2 Работа с упорядоченными данными

Все программы обрабатывают данные, и программы на Python — не исключение.

На самом деле *данные повсюду*. Ведь в основном программирование — это работа с данными: *получение* данных, *обработка* данных, *интерпретация* данных. Чтобы работать с данными более эффективно, нужен какой-то контейнер, куда их можно *сложить*. Python предоставляет удобные структуры данных *широкого применения*: **списки**, **словари**, **кортежи** и **множества**. В этой главе мы бегло рассмотрим все четыре, а затем углубимся в изучение **списков** (остальные три структуры подробнее рассмотрены в следующей главе). Мы уже затрагивали эту тему ранее, поскольку все, с чем нам приходится сталкиваться при программировании на Python, так или иначе относится к работе с данными.

Числа, строки... и объекты	84
Встречайте: четыре встроенные структуры данных	86
Словарь: неупорядоченная структура данных	88
Множество: структура данных, не позволяющая дублировать объекты	89
Создание литеральных списков	91
Если работаете с фрагментом кода большим, чем пара строк, используйте редактор	93
Заполнение списка во время выполнения	94
Проверка принадлежности с помощью in	95
Удаление объектов из списка	98
Добавление элементов в список	100
Вставка элементов в список	101
Как скопировать структуру данных	109
Списки расширяют нотацию с квадратными скобками	111
Со списками можно использовать диапазоны	112
Начало и конец диапазона в списках	114
Работаем со срезами в списке	116
Использование цикла «for» со списками в Python	122
Срезы в деталях	124
Когда не нужно использовать списки	127
Код из главы 2	128

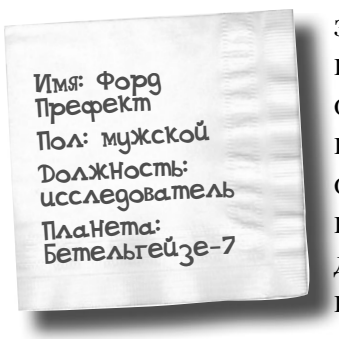
Структурированные данные

3 Работа со структурированными данными

Списки в Python очень удобны, но они не панацея.

Когда имеются *действительно* структурированные данные (для хранения которых список оказывается не лучшим выбором), спасение приходит от встроенных **словарей** Python. Словари «из коробки» позволяют хранить и обрабатывать данные, которые можно представить в виде *пар ключ/значение*. В этой главе мы изучим словари, а также **множества** и **кортежи**. Как и **списки** (которые мы изучили в главе 2), словари, множества и кортежи предоставляют встроенные инструменты для работы с данными, что делает Python еще более удобным языком.

Словари хранят пары ключ/значение 132
Как определяются словари в коде 134
Порядок добавления НЕ поддерживается 135
Выбор значений с помощью квадратных скобок 136
Работа со словарями во время выполнения программы 137
Изменение счетчика 141
Итерации по записям в словарях 143
Итерации по ключам и значениям 144
Итерации по словарям с использованием items 146
Насколько динамичны словари? 150
Предотвращение ошибок KeyError во время выполнения 152
Проверка вхождения с помощью in 153
Не забывайте инициализировать ключ перед использованием 154
Замена in на not in 155
Работа с методом setdefault 156
Создаем множества эффективно 160
Использование методов множеств 161
Сделаем пример с кортежами 168
Комбинирование встроенных структур данных 171
Доступ к данным, хранящимся в сложных структурах 177
Код из главы 3 179



4 Повторное использование

Функции и модули

Повторное использование кода — ключ к построению стабильных систем.

В случае Python все повторное использование начинается и заканчивается **функциями**. Возьмите несколько строк кода, дайте им имя — и у вас готовая функция (которую можно использовать повторно). Возьмите коллекцию функций и сохраните их в файле — у вас готовый **модуль** (который можно использовать повторно). Это правда, когда говорят, что *делиться приятно*, и в конце главы вы уже сможете создавать код для **многократного** и **совместного** использования, благодаря пониманию того, как работают функции и модули Python.

Повторное использование кода с помощью функций	182
Представляем функции	183
Вызываем функции	186
Функции могут принимать аргументы	190
Возврат одного значения	194
Возврат более одного значения	195
Вспомним встроенные структуры данных	197
Создание универсальной и полезной функции	201
Создание другой функции	202
Задание значений по умолчанию для аргументов	206
Позиционные и именованные аргументы	207
Повторим, что мы узнали о функциях	208
Запуск Python из командной строки	211
Создание необходимых файлов установки	215
Создание файла дистрибутива	216
Установка пакетов при помощи «pip»	218
Демонстрация семантики вызова по значению	221
Демонстрация семантики вызова по ссылке	222
Установка инструментов разработчика для тестирования	226
Соответствует ли наш код рекомендациям в PEP 8?	227
Разбираемся с сообщениями об ошибках	228
Код из главы 4	230

модуль

5 Построение веб-приложения

Возвращение в реальный мир

Вы уже знаете Python достаточно, чтобы быть опасными.

Четыре главы книги освоены, и сейчас вы в состоянии продуктивно использовать Python во многих областях применения (хотя многое еще предстоит узнать). Вместо того чтобы исследовать эти области, в этой и последующих главах мы изучим разработку веб-приложений — в этом Python особенно силен. Попутно вы еще больше узнаете о Python. Однако вначале позвольте кратко подытожить то, что вы уже знаете.

Python: что вы уже знаете 232
Чего мы хотим от нашего веб-приложения? 236
Давайте установим Flask 238
Как работает Flask? 239
Первый запуск веб-приложения Flask 240
Создание объекта веб-приложения Flask 242
Декорирование функции URL 243
Запуск функций веб-приложения 244
Размещение функциональности в Веб 245
Построение HTML-формы 249
Шаблоны связаны с веб-страничками 252
Отображение шаблонов из Flask 253
Отображение HTML-формы веб-приложения 254
Подготовка к запуску кода с шаблонами 255
Коды состояния HTTP 258
Обработка отправленных данных 259
Оптимизация цикла редактирование/остановка/запуск/проверка 260
Доступ к данным HTML-формы с помощью Flask 262
Использование данных запроса в веб-приложении 263
Выводим результат в виде HTML 265
Подготовка веб-приложения к развертыванию в облаке 274
Код из главы 5 277

6 Хранение и обработка данных

Где хранить данные

Рано или поздно появляется необходимость обеспечить надежное хранение данных.

И когда придет время **сохранить данные**, Python вам поможет. В этой главе вы узнаете о хранении и извлечении данных из *текстовых файлов*, которые — как механизм хранения — могут показаться слишком простыми, но тем не менее используются во многих проблемных областях. Кроме сохранения и извлечения данных из файлов, вы научитесь некоторым премудростям при работе с данными. «Серьезный материал» (хранение информации в базе данных) мы припасли для следующей главы, однако с файлами тоже придется потрудиться.

Работа с данными из веб-приложения	280
Python позволяет открывать, обрабатывать и закрывать	281
Чтение данных из существующего файла	282
Лучше «with», чем открыть, обработать, закрыть	284
Просмотр журнала в веб-приложении	290
Исследуем исходный код страницы	292
Пришло время экранировать (ваши данные)	293
Просмотр всего журнала в веб-приложении	294
Журналирование отдельных атрибутов веб-запроса	297
Журналирование данных в одну строку с разделителями	298
Вывод данных в читаемом формате	301
Генерируем читаемый вывод с помощью HTML	310
Встраиваем логику отображения в шаблон	311
Создание читаемого вывода с помощью Jinjia2	312
Текущее состояние кода нашего веб-приложения	314
Задаем вопросы о данных	315
Код из главы 6	316

Form Data	Remote_addr	User_agent	Results
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36	{'e', 'i'}

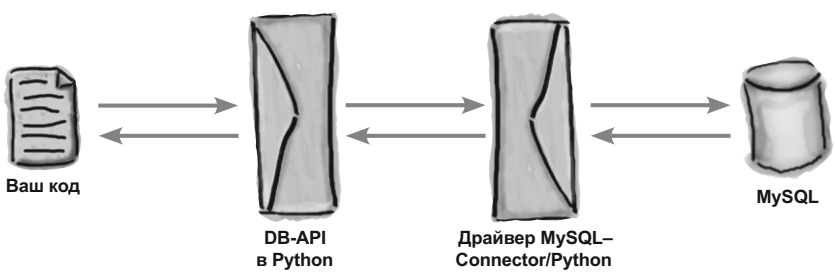
Использование базы данных

7 Используем DB-API в Python

Хранить информацию в реляционной базе данных очень удобно.

В этой главе вы узнаете, как организовать взаимодействие с популярной базой данных (БД) **MySQL**, используя универсальный прикладной программный интерфейс, который называется **DB-API**. Интерфейс DB-API (входящий в состав стандартной библиотеки Python) позволяет писать код, не зависящий от конкретной базы данных… если база данных понимает SQL. Хотя мы будем использовать MySQL, ничто не помешает вам использовать код DB-API с вашей любимой реляционной базой данных, какой бы она ни была. Давайте посмотрим, как пользоваться реляционной базой данных в Python. В этой главе не так много нового в плане изучения Python, но использование Python для общения с БД — **это важно**, поэтому стоит поучиться.

Включаем поддержку баз данных в веб-приложении	318
Задача 1. Установка сервера MySQL	319
Введение в Python DB-API	320
Задача 2. Установка драйвера базы данных MySQL для Python	321
Установка MySQL–Connector/Python	322
Задача 3. Создание базы данных и таблиц для веб-приложения	323
Выбираем структуру для журналируемых данных	324
Убедимся, что таблица готова к использованию	325
Задача 4. Программирование операций с базой данных и таблицами	332
Хранение данных — только половина дела	336
Как организовать код для работы с базой данных?	337
Подумайте, что вы собираетесь использовать повторно	338
А что с тем импортом?	339
Вы видели этот шаблон раньше	341
Неприятность не такая уж неприятная	342
Код из главы 7	343



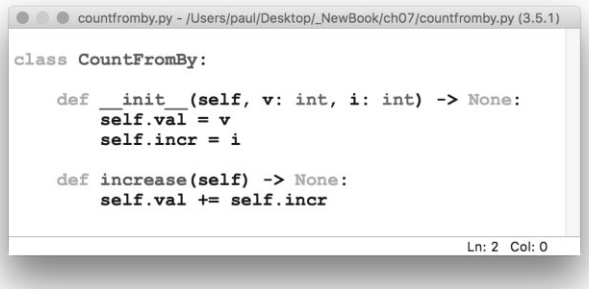
8 Немного о классах

Абстракция поведения и состояния

Классы позволяют связать поведение кода и состояние вместе.

В этой главе мы отложим веб-приложение в сторону и будем учиться создавать **классы**. Это умение понадобится вам при создании диспетчера контекста. Классы — настолько полезная штука, что вам в любом случае стоит ознакомиться с ними поближе, поэтому мы посвятили им отдельную главу. Мы не будем рассматривать классы во всех подробностях, а коснемся лишь тех аспектов, которые пригодятся для создания диспетчера контекста, которого ожидает наше веб-приложение. А теперь вперед, посмотрим, что к чему.

Подключаемся к инструкции «with»	346
Объектно-ориентированный пример	347
Создание объектов из классов	348
Объекты обладают общим поведением, но не состоянием	349
Расширяем возможности CountFromBy	350
Вызов метода: подробности	352
Добавление метода в класс	354
Важность «self»	356
Область видимости	357
Добавляйте к именам атрибутов приставку «self»	358
Инициализация атрибута перед использованием	359
Инициализация атрибутов в «init» с двойными подчеркиваниями	360
Инициализация атрибутов в «__init__»	361
Представление CountFromBy	364
Определение представления CountFromBy	365
Определение целесообразных умолчаний для CountFromBy	366
Классы: что мы знаем	368
Код из главы 8	369



Протокол управления контекстом

9 Подключение к инструкции with

Пора применить все изученное на практике.

В главе 7 мы обсудили использование **реляционных баз данных** в Python, а в главе 8 ознакомились с использованием **классов**. В главе 9 мы объединим обе методики и создадим **диспетчер контекста**, который позволит расширить инструкцию `with` для работы с системами реляционных баз данных. В этой главе вы подключитесь к инструкции `with`, создав новый класс, соответствующий требованиям **протокола управления контекстом** в языке Python.

Выбираем лучший способ повторного использования кода для работы с БД 372
Управление контекстом с помощью методов 374
Вы уже видели, как действует диспетчер контекста 375
Создание нового класса диспетчера контекста 376
Инициализация класса параметрами соединения с базой данных 377
Выполняем настройку в «__enter__» 379
Выполнение завершающего кода с использованием «__exit__» 381
Повторное обсуждение кода веб-приложения 384
Вспомним функцию «log_request» 386
Изменение функции «log_request» 387
Вспомним функцию «view_the_log» 388
Изменился не только этот код 389
Изменение функции «view_the_log» 390
Ответы на вопросы о данных 395
Код из главы 9 396


```
File  Edit  Window  Help  Checking our log DB
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+-------------------------+---------+-----------+----------------+-----------------+
| id | ts                  | phrase                  | letters | ip        | browser_string | results         |
+----+---------------------+-------------------------+---------+-----------+----------------+-----------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker             | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}      |
|  3 | 2016-03-09 13:42:15 | galaxy                  | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}      |
|  4 | 2016-03-09 13:43:07 | hitch-hiker             | xyz     | 127.0.0.1 | firefox        | set()           |
+----+---------------------+-------------------------+---------+-----------+----------------+-----------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

10 Декораторы функций

Обертывание функций

Протокол управления контекстом из главы 9 — не единственная возможность улучшить код.

Python также позволяет использовать **декораторы** функций — технологию, с помощью которой можно добавлять код к существующим функциям, *не* изменяя имеющийся код. Если вы увидите в этом сходство с черной магией, не переживайте: ничего подобного. Однако многие программисты на Python считают создание декораторов функций одной из наиболее сложных сторон языка, поэтому пользуются этой возможностью реже, чем должны бы. В этой главе мы покажем, что создавать и использовать декораторы совсем не сложно, несмотря на всю их «продвинутость»

Веб-сервер (не ваш компьютер) запускает ваш код	402
Поддержка сеансов в Flask позволяет хранить состояние	404
Поиск по словарю позволяет получить состояние	405
Организация входа в систему с помощью сеансов	410
Выход из системы и проверка состояния	413
Передаем функцию в функцию	422
Вызываем переданную функцию	423
Принимаем список аргументов	426
Обработка списка аргументов	427
Принимаем словарь аргументов	428
Обработка словаря аргументов	429
Принимаем любое количество аргументов любого типа	430
Создание декоратора функции	433
Последний шаг: работа с аргументами	437
Использование декоратора	440
Назад к ограничению доступа к /viewlog	444
Код из главы 10	446

11 Обработка исключений

Что делать, когда что-то идет не так

Каким бы хорошим ни был ваш код, иногда все равно что-то идет не так.

Вы успешно выполнили все примеры в книге и, скорее всего, убедились, что имеющийся код работает. Значит ли это, что его можно считать надежным? По всей видимости, нет. Писать код и надеяться, что ничего плохого не случится, в лучшем случае наивно. В худшем случае — опасно, поскольку что-то непредвиденное порой все же происходит (и будет происходить). При написании кода лучше быть осторожным, а не доверчивыми. Осторожным, чтобы ваш код делал именно то, что вам нужно, и правильно реагировал, если что-то пойдет не так. В этой главе вы увидите, что может пойти не так, и узнаете, что делать, когда (и, чаще всего, прежде чем) такое произойдет.

Базы данных не всегда доступны 454
Веб-атаки — настоящая боль 455
Ввод-вывод бывает медленным (иногда) 456
Вызовы функций могут оканчиваться неудачей 457
Всегда используйте try для кода с высоким риском ошибок 459
Одна инструкция try, но несколько except 462
Обработчик любых исключений 464
Узнаем об исключениях из «sys» 466
Универсальный обработчик, повторение 467
Назад к нашему веб-приложению 469
Тихая обработка исключений 470
Обработка других ошибок в базе данных 476
Избегайте тесных связей в коде 478
Модуль DBcm, еще раз 479
Создание собственного исключения 480
Что еще может пойти не так с «DBcm»? 484
Обработка SQLError отличается 487
Возбуждаем SQLError 489
Подведение итогов: добавление надежности 491
Как быть с ожиданием? Есть варианты... 492
Код из главы 11 493


```
   ...
Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |     +-- FloatingPointError
      |     +-- OverflowError
      |     +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
            ...
```

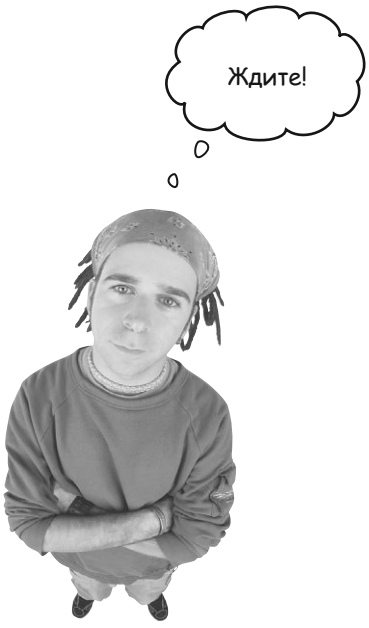
Немного о многопоточности

11 3/4 Обработка ожидания

Иногда для выполнения кода может потребоваться много времени.

В одном случае это становится проблемой, в другом — нет. Если код работает «за кулисами» и ему нужно 30 секунд, чтобы сделать какие-то свои дела, ожидание не превращается в проблему. Однако если приложению требуется 30 секунд, чтобы ответить пользователю, это заметят все. Решение проблемы зависит от того, какие действия пытаетесь выполнить вы (и кто именно ожидает). В этой короткой главе мы обсудим некоторые варианты, а затем рассмотрим одно из решений данной проблемы: *как быть, если что-то требует слишком много времени?*

Ожидание: что делать?	498
Как выполняются запросы к базе данных?	499
Инструкции INSERT и SELECT базы данных отличаются	500
Делаем несколько дел сразу	501
Не грустим: используем потоки	502
В первую очередь: не паниковать	506
Не грустим: Flask нам поможет	507
Надежно ли ваше веб-приложение сейчас?	510
Код из главы 11¾	511



12 Продвинутые итерации

Безумные циклы

Удивительно, сколько времени порой наши программы проводят внутри циклов.

Ни для кого не секрет, что большинство программ предназначено для быстрого выполнения одинаковых действий много-много раз. Говоря об оптимизации циклов, можно выделить два подхода: (1) улучшение синтаксиса (чтобы упростить определение циклов) и (2) улучшение способа выполнения (чтобы ускорить работу циклов). Во времена Python 2 (то есть давным-давно) создатели языка добавили единственную языковую возможность, которая реализует оба подхода и называется довольно странно — **генераторы**. Но пусть это название вас не пугает. К концу главы вы будете удивляться, как вам вообще удавалось столько времени обходиться без генераторов.

Чтение данных CSV в виде списков 515
Чтение CSV-данных в словарь 516
Удаление пробелов и разбивка исходных данных 518
Будьте внимательны при составлении цепочек из вызовов методов 519
Преобразование данных в нужный формат 520
Преобразование в словарь списков 521
Шаблон со списками 526
Преобразование шаблонного кода в генераторы 527
Подробнее о генераторах 528
Определение генератора словарей 530
Расширенные генераторы с фильтрами 531
Преодолеваем сложности, как это принято в Python 535
Генератор множеств в действии 541
А что с «генераторами кортежей»? 543
Круглые скобки вокруг кода == Генератор 544
Использование генератора списков для обработки URL 545
Использование выражения-генератора для обработки URL 546
Определим, что должна делать функция 548
Ощути силу функций-генераторов 549
Исследуем функцию-генератор 550
Последний вопрос 554
Код из главы 12 555
Пора прощаться... 556

Установка

a

Установка Python

Начнем сначала: установим Python на компьютер.

Какой бы ОС вы ни пользовались — *Windows*, *Mac OS X* или *Linux*, — Python для вас доступен. Порядок установки на каждую из этих платформ зависит от организации процесса в каждой из операционных систем (мы знаем… это звучит пугающе, да?), но разработчики Python хорошо потрудились, создав мастер установки для каждой из популярных систем. В этом коротком приложении вы узнаете, как установить Python на компьютер.

Установка Python 3 в Windows 558
Проверим правильность установки Python 3 в Windows 559
Расширяем набор инструментов Python 3 в Windows 560
Установка Python 3 в Mac OS X (macOS) 561
Проверка и настройка Python 3 в Mac OS X 562
Установка Python 3 в Linux 563

PythonAnywhere

b

Развертывание веб-приложения

В конце главы 5 мы говорили, что развертывание веб-приложения в облаке займет всего лишь 10 минут.

Пора выполнить обещание. В этом приложении мы собираемся провести вас через весь процесс развертывания веб-приложения в *PythonAnywhere* за 10 минут с самого начала до полностью развернутого приложения. *PythonAnywhere* — это служба, популярная в сообществе программирования на Python, и нетрудно понять почему: она работает в полном соответствии с ожиданиями, имеет отличную поддержку Python (и Flask) и — самое важное — позволяет начать с бесплатного размещения веб-приложения. Давайте оценим *PythonAnywhere*.

Шаг 0: Небольшая подготовка 566
Шаг 1: Регистрируемся в PythonAnywhere 567
Шаг 2: Выгружаем файлы в облако 568
Шаг 3: Извлекаем и устанавливаем код 569
Шаг 4: Создаем начальное веб-приложение, 1 из 2 570
Шаг 4: Создаем начальное веб-приложение, 2 из 2 571
Шаг 5: Настраиваем веб-приложение 572
Шаг 6: Запускаем облачное веб-приложение! 573

Топ-10 тем, которые мы не рассмотрели

Всегда есть чему поучиться

Мы и не стремились рассмотреть все.

Наша цель — познакомить вас с Python, чтобы вы как можно скорее могли приступить к работе. Мы о многом умолчали. В этом приложении мы обсудим топ-10 тем, которые — будь у нас еще 600 страниц или около того — мы не обошли бы вниманием. Не все 10 пунктов будут вам интересны, но вы можете быстро пролистать их — вдруг мы затронули интересную для вас тему или дали ответ на насущный вопрос. Все технологии программирования в приложении готовы к использованию в Python и его интерпретаторе.

1. Что насчет Python 2? 576
2. Виртуальное программное окружение 577
3. Больше объектной ориентированности 578
4. Форматирование строк и тому подобное 579
5. Сортировка 580
6. Больше из стандартной библиотеки 581
7. Параллельное выполнение кода 582
8. Графический интерфейс с использованием Tkinter (и веселье с turtle) 583
9. Работа не завершена, пока не проверена 584
10. Отладка, отладка, отладка 585

Д Топ-10 проектов, которые мы не рассмотрели

Еще больше инструментов, библиотек и модулей

Мы знаем, о чем вы подумали, читая заголовок этого Приложения.

Почему они не назвали предыдущее приложение: *«Топ-20 тем, которые не были рассмотрены»*? Зачем нам еще 10? В предыдущем приложении мы ограничили обсуждение инструментами, которые поставляются вместе с Python (те самые «батарейки», которые входят в комплект). В этом приложении мы пойдем дальше и обсудим группу технологий, которые стали доступны, *потому что* существует Python. Здесь перечислено много полезного, но, как в случае с предыдущим приложением, если вы бегло просмотрите оставшиеся страницы, то не пропустите *ничего важного*.

1. Альтернативы командной строке >>> 588
2. Альтернативы интегрированной среде IDLE 589
3. Jupyter Notebook: IDE на основе веб 590
4. Наука работы с данными 591
5. Технологии веб-разработки 592
6. Работа с данными в веб 593
7. Еще больше источников данных 594
8. Инструменты для программирования 595
9. Kivy: наш выбор в номинации «Лучший проект всех времен» 596
10. Альтернативные реализации 597

Присоединяйтесь

Сообщество Python

Python — это гораздо больше, чем отличный язык программирования.

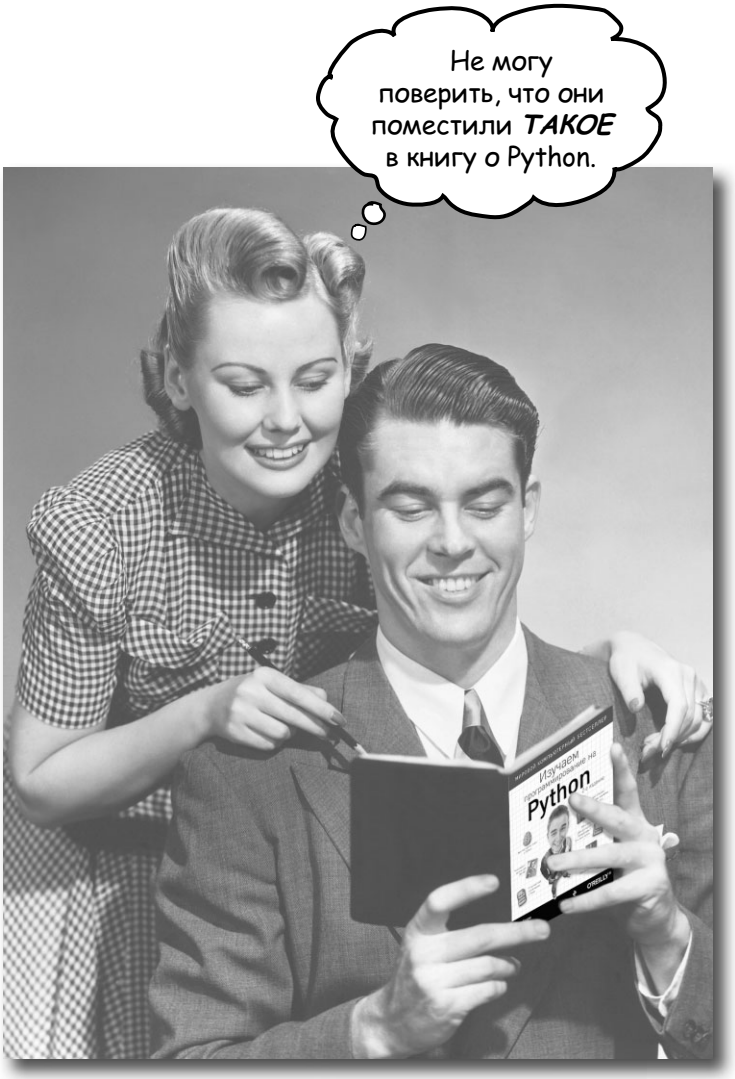
Это отличная компания. Сообщество Python гостеприимное, разнообразное, открытое, дружественное, щедрое и готовое делиться. Просто удивительно, что еще никто не напечатал такую поздравительную открытку! Если серьезно, то Python — не столько язык, сколько особая наука программирования. Вокруг Python сформировалась целая экосистема: отличные книги, блоги, веб-сайты, конференции, встречи, группы пользователей и великие личности. В этом приложении мы познакомимся с сообществом Python и посмотрим, что оно может предложить. Не пишите код в одиночестве: **присоединяйтесь к сообществу!**

BDFL: Benevolent Dictator for Life — великодушный пожизненный диктатор 600
Толерантное сообщество: уважение к многообразию 601
Подкасты Python 602
Дзен Python 603
Дзен Python от Тима Петерса 604

Алфавитный указатель 605

Как читать эту Книгу

Введение



В этом разделе мы отвечаем на животрепещущий вопрос: «Зачем они поместили ТАКОЕ в книгу о Python?»

Для кого эта книга?

Если вы утвердительно ответите на все эти вопросы:

1. Вы уже знаете, как программировать на другом языке программирования?
2. Вы хотите иметь пошаговое руководство для программирования на Python, добавить его в свой список инструментов и делать с его помощью что-то новое?
3. Вы предпочитаете делать что-то новое и получать знания по ходу дела, а не слушать лекции в течение нескольких часов подряд?

тогда эта книга для вас.

Это НЕ справочное руководство, и мы предполагаем, что вы ранее программировали.

Кому не стоит читать эту книгу?

Если вы утвердительно ответите на любой из этих вопросов:

1. Вы уже знаете большинство из того, что необходимо для программирования на Python?
2. Вы ищете справочник по Python, охватывающий все детали языка в мельчайших подробностях?
3. Вы скорее предпочтете подвергнуться пыткам, чем изучать что-то новое? Вы считаете, что книга о Python должна охватывать все, и если она своей скукой доведет читателя до слез, то это даже лучше?

тогда эта книга ***не*** для вас.

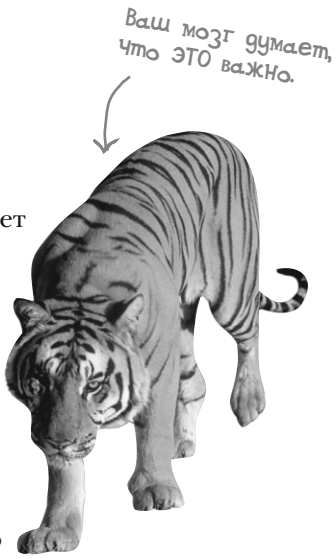
Заметка отдела продаж: эта книга для всех, у кого есть деньги...

Мы знаем, о чем вы подумали

«Как *эта* книга может считаться серьезной книгой о Python?»

«Что тут еще за картинки?»

«Неужели я действительно смогу *учиться* таким способом?»



Мы знаем, о чем подумал ваш *мозг*

Ваш мозг жаждет нового. Это всегда поиск, наблюдение, *ожидание* чего-то необычного. Он так устроен, и это помогает вам остаться в живых.

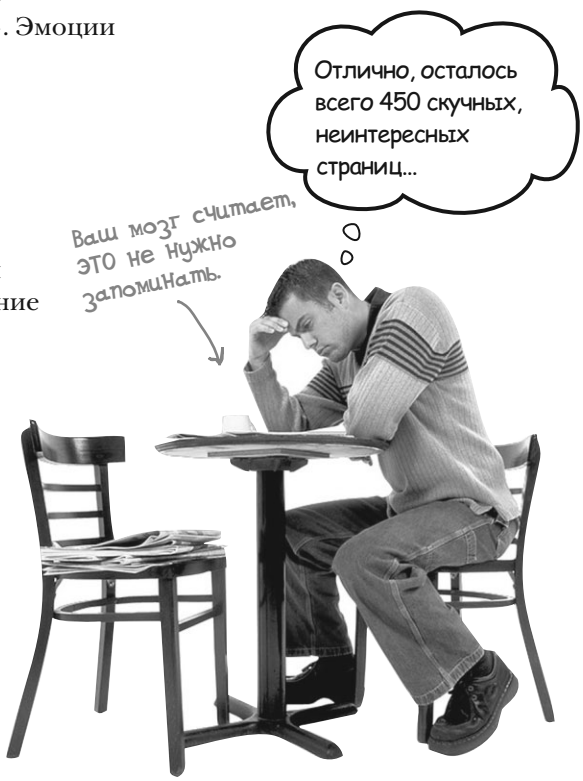
Итак, что же ваш мозг будет делать со всеми рутинными, обычными, нормальными вещами, с которыми он столкнется? Он сделает все *возможное*, чтобы помешать им вмешиваться в *реальную* работу мозга — фиксировать действительно *важные* события. Мозг никогда не запоминает скучные факты — они никогда не пройдут фильтр «очевидно, это не важно».

Как ваш мозг *узнает*, что именно важно? Предположим, что вы пошли в поход на день и на вас набросился тигр. Что произойдет в вашей голове и теле? Нейроны «зажгутся». Эмоции взорвутся. Произойдет *выброс химических веществ*.

Вот откуда мозг узнает...

Это должно быть важно! Не забывай об этом!

Теперь представьте, что вы находитесь дома или в библиотеке. Это безопасное, теплое место, где нет никаких тигров. Вы учитесь. Готовитесь к экзамену. Или пытаетесь освоить какую-то техническую тему, на изучение которой, как думает ваше начальство, требуется неделя, максимум — 10 дней.



Есть только одна проблема. Ваш мозг пытается сделать вам одолжение. Он пытается убедиться, что эта очевидно неважная информация не будет расходовать дефицитные ресурсы. Ресурсы, которые лучше потратить на запоминание действительно важных вещей. Вроде тигров, угрозы пожара или размещения фотографии с вечеринки на своей страничке в «Фейсбуке». И нет никакого простого способа сказать вашему мозгу: «Эй, мозг, большое спасибо, но несмотря на то, что эта книга скучная и вызывает у меня ноль эмоций по шкале Рихтера, я действительно хочу, чтобы ты все это запомнил».

Мы думаем о читателе этой книги, как о человеке, способном учиться

Так что же нужно, чтобы *научиться* чему-то? Во-первых, нужно *усвоить* информацию, а потом убедиться, что вы не *забудете* ее. Это не заталкивание фактов в вашу голову. Основываясь на последних исследованиях в области когнитивной науки, нейробиологии и образовательной психологии, *обучение* — это нечто намного большее, чем чтение текста на странице. Мы знаем, что «включает» ваш мозг.

Некоторые принципы обучения способом «быстрого вхождения»:

Визуализация. Изображения запоминаются гораздо лучше, чем одни только слова, и делают обучение более эффективным (способность запоминать и передавать знания возрастает до 89%). Такой подход делает информацию более понятной. Если **слова размещены внутри или возле изображений**, к которым они относятся, а не под ними или на другой странице, вероятность успешного решения задачи учениками увеличивается *в два раза*.

Разговорный стиль и персонификация. Новейшие исследования показали, что если материал изложен не формальным языком, а разговорным, с повествованием от первого лица, то учащиеся справляются с итоговыми тестами на 40% лучше. Вместо чтения лекций рассказывайте истории. Используйте повседневный язык. Не напускайте на себя излишнюю серьезность. Чему бы *вы* уделили больше внимания: интересной беседе за ужином с друзьями или лекции?

Глубокое погружение обучаемого в материал. Другими словами, пока вы не заставите нейроны активно работать, в вашей голове ничего происходить не будет. Читатель должен быть мотивирован, заинтересован, заинтригован и вдохновлен, чтобы решать проблемы, делать выводы и генерировать новые знания. И для этого нужны задачи, упражнения и наводящие на размышления вопросы, а также деятельность, которые включают оба полушария мозга и вызывают эмоции.

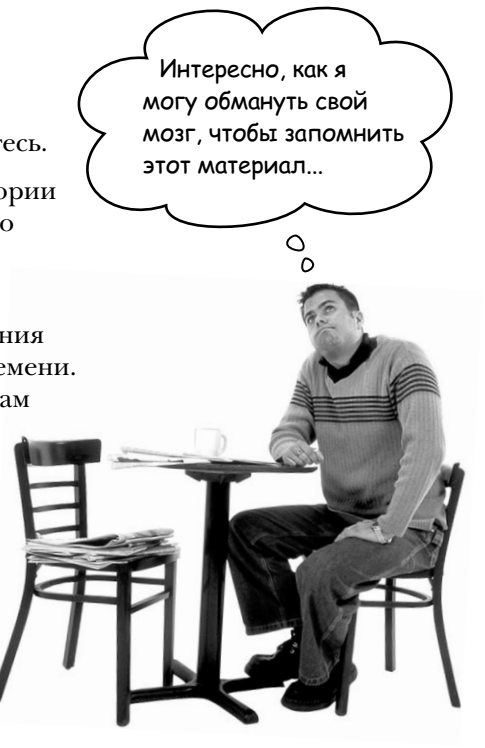
Захват и удержание внимания читателя. У каждого из нас было такое «Я действительно хочу выучить это, но мне хочется спать уже после первой страницы». Мозг обращает внимание на все необычное: интересное, странное, броское, неожиданное. Изучение новой сложной технической темы не должно быть скучным. Мозг будет учиться гораздо быстрее, если ему интересно.

Эмоции. Теперь мы знаем, что способность к запоминанию во многом зависит от эмоционального наполнения. Вы запоминаете то, что для вас важно. Вы запоминаете, когда вы что-то чувствуете. Нет, мы не имеем в виду душераздирающую историю о мальчике и его собаке. Мы говорим о таких эмоциях, как удивление, любопытство, развлечение, «а что, если …?» и, конечно же, «да я крутой!», которые возникают, когда вы решаете головоломку, понимаете что-то такое, что все остальные считают трудным, или знаете нечто, что позволяет считать себя «более подкованным технически», чем Боб из технического отдела, который этого не знает.

Метапознание: размышления о мышлении

Если вы действительно хотите учиться и хотите учиться быстрее и основательнее, обратите внимание на то, как вы обращаете внимание. Подумайте о том, как вы думаете. Изучите, как вы учитесь.

Большинство из нас не проходили курсы по метапознанию и теории обучения, пока мы росли. Мы *должны* были учиться, но нас редко *учили* учиться.



Но мы предполагаем, что, если вы держите эту книгу, вы действительно хотите научиться решать задачи программирования с помощью Python. И вы, вероятно, не хотите тратить много времени. Если вы хотите использовать то, что прочитаете в этой книге, вам нужно *запоминать* то, что читаете. Для этого вы должны *понимать*. Чтобы получить максимальную пользу от этой или *любой* другой книги или опыта обучения, возьмите на себя ответственность за свой мозг. За *наполнение* мозга.

Хитрость заключается в том, чтобы заставить мозг считать новый материал, который вы изучаете, Действительно Очень Важным. Имеющим решающее значение для вашего благополучия. Таким же важным, как тигр. В противном случае вы будете постоянно воевать со своим мозгом, и он сделает все возможное, чтобы не сохранять новую информацию.

КАК заставить свой мозг воспринимать Python так, словно это не язык программирования, а голодный тигр?

Есть два способа: медленный и утомительный или быстрый и эффективный. Медленный — это способ простого повторения. Вы, очевидно, знаете, что *можно* выучить и запомнить даже скучную тему, если продолжать вбивать одно и то же в свой мозг. После достаточного количества повторений ваш мозг скажет: «*Не похоже*, что это для него важно, но раз он продолжает твердить это *снова*, и *снова*, и *снова*, то, по-видимому, придется подчиниться».

Более быстрый способ — ***сделать что-нибудь, что повысит активность мозга***, подключая различные *типы* активности мозга. Описанное на предыдущей странице является значительной частью решения. Доказано, что все эти факторы помогают мозгу работать вам во благо. Например, исследования показывают, что *размещение* слов в изображениях, которые они описывают (а не в подписи, в тексте или где-нибудь еще на странице), заставляет ваш мозг построить между словами и картинками смысловые связи, что приводит к увеличению количества задействованных нейронов. Больше задействованных нейронов — больше шансов, что ваш мозг *поймет*: на это стоит обратить внимание и, возможно, запомнить.

Разговорный стиль помогает, потому что люди, как правило, уделяют больше внимания содержимому, когда они чувствуют, что участвуют в разговоре, поскольку разговор требует удержания внимания на всей его протяженности до самого конца. Удивительно, но ваш мозг не *волнует*, что «разговор» ведется между вами и книгой! С другой стороны, если стиль письма формальный и сухой, ваш мозг воспринимает это точно так же, как чтение лекции, когда вы сидите в аудитории, заполненной пассивными участниками. Тут даже бодрствовать нет необходимости.

Но изображения и разговорный стиль — это только начало...

Вот что мы сделали:

Мы использовали ***картинки***, потому что мозг воспринимает зрительные образы лучше, чем текст. Там, где речь идет о мозге, одна картинка действительно *стоит* тысячи слов. И когда текст и изображения работают вместе, мы *встраиваем* текст в изображения, потому что ваш мозг работает более эффективно, когда текстовое описание находится в пределах образа, к которому он относится, в отличие от случая, когда текст находится в заголовке или спрятан где-то глубоко в тексте.

Мы использовали ***избыточность***, повторяя одно и то же *по-разному*, используя *множественные способы* представления информации, чтобы увеличить вероятность сохранения содержимого в нескольких областях мозга.

Мы использовали понятия и изображения ***неожиданными*** способами, потому что мозг настроен на новизну. Мы использовали изображения и понятия, имеющие хоть какое-то, по крайней мере, *некоторое* ***эмоциональное*** *содержание*, потому что ваш мозг настроен обращать внимание на биохимию эмоций. То, что заставляет вас *чувствовать* что-то, скорее всего, следует запомнить, даже если это чувство — лишь ***улыбка***, ***удивление*** или ***любопытство***.

Мы использовали более личный ***разговорный стиль***, потому что мозг настроен уделять больше внимания, когда считает, что вы участвуете в разговоре, а не пассивно слушаете презентацию. Ваш мозг делает это даже тогда, когда вы *читаете*.

Мы включили более 80 ***практических упражнений***, потому что мозг настроен учиться и запоминать больше, когда вы ***делаете*** что-либо, а не просто *читаете* об этом. И мы сделали упражнения сложными, но выполнимыми, потому что это именно то, что предпочитает большинство людей.

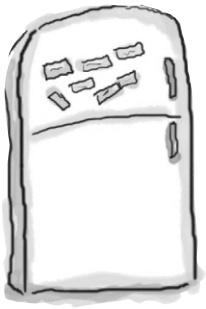
Мы использовали ***несколько стилей подачи материала***, потому что одни предпочитают учиться постепенно, «шаг за шагом», другие желают сначала понять общую картину, а третьи просто хотят увидеть примеры. Независимо от ваших собственных предпочтений в обучении, будет полезным увидеть один и тот же материал, представленный несколькими способами.

Мы подготовили материал для ***обоих полушарий вашего мозга***, потому что чем больше участков мозга вы задействуете, тем лучше пройдет обучение и запоминание и тем дольше вы можете оставаться сосредоточенным. Использование одного полушария мозга означает, что другое полушарие в это время отдыхает, так что ваше обучение будет продуктивным в течение длительного периода времени.

Также мы включили в книгу ***рассказы*** и упражнения, которые представляют ***более одной точки зрения***, потому что мозг настроен на более глубокое изучение, когда вынужден давать оценки и выстраивать суждения.

Мы включили ***задачи***, упражнения и ***вопросы***, на которые не всегда есть однозначный ответ, потому что мозг настроен учиться и запоминать, когда он работает над чем-то. Подумайте об этом: чтобы поддерживать хорошую физическую *форму*, недостаточно просто *наблюдать* за людьми в тренажерном зале. Но мы сделали все возможное, чтобы убедиться, что когда вы усердно трудитесь, то находитесь на *верном* пути. Вы ***не потратите ни одного лишнего дендрита*** на изучение трудного для понимания примера или разбор трудного, насыщенного терминологией или чрезмерно лаконичного текста.

В книге есть ***люди***. В рассказах, примерах, рисунках и так далее, просто потому что *вы* человек. И ваш мозг уделяет *людям* больше внимания, чем *вещам*.



Вырежьте это и приклейте На холодильник.

Вот как ВЫ можете заставить свой мозг работать

Итак, с нашей стороны мы сделали всё. Остальное зависит от вас. Эти советы являются отправной точкой. Прислушивайтесь к своему мозгу и выясняйте, что для вас работает, а что нет. Попробуйте новые приемы.

1 Не торопитесь. Чем больше вы поймете, тем меньше вам надо будет запоминать.

Недостаточно просто *читать*. Остановитесь и подумайте. Когда в книге задается вопрос, не переходите сразу к ответу. Представьте, что кто-то действительно задает вам вопрос. Чем глубже вы заставляете свой мозг вникать в материал, тем больше шансов научиться и запомнить.

2 Выполняйте задания. Пишите свои заметки.

Мы предоставили вам задания, но если бы мы решили их за вас, это было бы все равно что кто-то другой ходит на тренировки вместо вас. И не просто смотрите на упражнения. **Пользуйтесь карандашом.** Существует множество доказательств, что физическая активность *в процессе* занятий повышает качество обучения.

3 Читайте разделы «Это не глупые вопросы».

То есть читайте их все. Это не дополнительные врезки, ***они являются частью основного материала!*** Не пропускайте их.

4 Пусть книга станет последней прочитанной перед сном. По крайней мере не читайте после ничего сложного.

Часть обучения (особенно перенос в долговременную память) происходит *после* того, как вы закрыли книгу. Вашему мозгу требуется время на собственную, дополнительную обработку. Если вы загрузите его чем-то новым во время этой обработки, часть того, что вы только что узнали, будет утрачена.

5 Проговаривайте прочитанное. Вслух.

Разговор активирует другую часть мозга. Если вы пытаетесь понять что-то или увеличить свой шанс вспомнить это позже, произнесите это вслух. А еще лучше, попробуйте объяснить это вслух кому-то другому. Вы научитесь гораздо быстрее, и у вас могут появиться идеи, которых не возникало, когда вы читали книгу.

6 Пейте воду. Много воды.

Мозг работает лучше всего, когда в организме достаточно жидкости. Обезвоживание (которое может наступить прежде, чем вы почувствуете жажду) уменьшает способность к познанию.

7 Прислушивайтесь к своему мозгу.

Следите за тем, чтобы мозг не перегружался. Если вы почувствовали, что перестали воспринимать текст или начали забывать то, что только что прочитали, пришло время сделать перерыв. Достигнув определенного момента, вы не сможете обучаться быстрее, пытаясь запомнить еще больше, а только навредите процессу обучения.

8 Чувствуйте что-нибудь.

Ваш мозг должен знать, что обучение имеет для вас значение. Вовлекайтесь в истории. Составьте собственные подписи для изображений. Вздыхать от плохой шутки всё равно лучше, чем вообще ничего не чувствовать.

9 Пишите много кода!

Есть только один способ научиться программировать на Python: **писать много кода**. И это то, что вы должны делать на протяжении всей книги. Программирование — это навык, и единственный способ развить этот навык — практика. Мы дадим вам достаточно много практических заданий: в каждой главе есть упражнения, которые предлагают вам решить задачу. Не пропускайте их: большая часть обучения происходит именно тогда, когда вы выполняете упражнения. Мы приводим решение каждого задания — **не бойтесь заглянуть туда**, если застряли! (Очень легко застрять из-за какой-то мелочи.) Но попытайтесь решить эту задачу самостоятельно, прежде чем подсмотрите решение. И, конечно, заставьте его работать, прежде чем продолжить читать.

Прочти меня, 1 из 2

Это обучающий практикум, а не справочник. Мы сознательно убрали все, что может помешать изучению материала. Проходя обучение в первый раз, читайте все с самого начала, поскольку мы уже учли все, что вы могли видеть и знать до этого.

Эта книга создана для того, чтобы вы обучались как можно быстрее.

Мы обучим только тому, что вам необходимо знать. Поэтому вы не найдете здесь ни длинных списков технических сведений, ни таблиц операторов Python, ни правил приоритета. Мы не рассматриваем *все и вся*, но мы упорно трудились, чтобы охватить самое основное как можно лучше, и теперь вы сможете *быстро* «загрузить» Python в свой мозг и удержать его там. Единственное предположение, которое мы делаем, — что вы уже знаете, как программировать на другом языке программирования.

Эта книга ориентирована на Python 3.

В этой книге мы используем версию 3 языка программирования Python. Мы рассмотрим, как получить и установить Python 3 в *Приложении А*. В этой книге **не** используется Python 2.

Мы сразу начнем работу с Python.

С первой главы мы начнем получать полезные навыки и в дальнейшем будем на них опираться. Нет смысла ходить вокруг да около, потому что мы хотим, чтобы вы сразу начали *продуктивно* использовать Python.

Практические задания НЕ факультативны, вы должны выполнять их.

Упражнения и практические задания не являются дополнительными, это часть основного материала книги. Некоторые нужны, чтобы помочь запоминанию, некоторые важны для понимания, а некоторые помогут вам применять то, что вы узнали. ***Не пропускайте упражнения.***

Избыточность в книгу внесена умышленно и является важной.

Важной отличительной особенностью книги является наше искреннее желание, чтобы вы *действительно* усвоили материал. И мы хотим, чтобы вы, прочитав книгу, помнили то, что узнали. Большинство справочников не ставят целью сохранение и удержание информации, но эта книга для обучения. И вы увидите, что некоторые понятия встречаются больше одного раза.

Примеры упрощены, насколько это возможно.

Наши читатели говорят нам, что их разочаровывает, когда приходится пробираться через 200 строк примера и искать две строчки кода, которые нужно понять. Большинство примеров в этой книге представлены в минимально возможном контексте, так что та часть, которую вы пытаетесь выучить, будет ясной и простой. Не ожидайте, что все примеры будут устойчивыми или даже полными — они написаны специально для обучения и не всегда полностью функциональны (хотя мы стараемся это обеспечить, насколько возможно).

Прочти меня, 2 из 2

Да, еще немного...

Это второе издание, и оно не точно такое же, как первое.

Это обновление первого издания, опубликованного в конце 2010 года. Несмотря на то что автор тот же, теперь он стал старше и (надеюсь) мудрее, поэтому было решено полностью переписать материал первого издания. Так что... здесь *все* новое: порядок изложения отличается, содержание было обновлено, примеры стали лучше, истории либо исчезли совсем, либо изменены. Мы сохранили обложку с незначительными поправками, так как решили не раскачивать лодку слишком сильно. Прошло долгих шесть лет... мы надеемся, что вам понравится то, что мы придумали.

Где код?

Мы разместили примеры кода на веб-сайтах, чтобы вы могли копировать и вставлять их по мере необходимости (хотя мы рекомендуем вводить код вручную по мере изучения материала). Вы найдете код по ссылкам:

https://eksmo.ru/files/Barry_Python_Head_First.zip

Команда технических редакторов

Билл Любанович (Bill Lubanovic) занимается программированием и администрированием уже сорок лет. Также он пишет для издательства O'Reilly: написал главы двух книг по безопасности Linux, соавтор книги для администраторов Linux и автор книги «Introducing Python» («Простой Python. Современный стиль программирования»). Он живет у замерзшего озера в горах Сангре-де-Сасквоч Миннесоты с любимой женой, двумя прекрасными детьми и тремя пушистыми кошками.

Эдвард Ю Шунг Вонг (Edward Yue Shung Wong) увлечен программированием с тех пор, как написал свою первую строчку кода на Haskell в 2006 году. В настоящее время работает над событийно-ориентированной обработкой процесса торгов в самом центре Лондона. Он с удовольствием разделяет страсть к разработке с Java Community London и Software Craftsmanship Community. Если Эдвард не за клавиатурой, значит его можно найти на футбольном поле или играющим на YouTube (@arkangelofkaos).

Адрианна Лов (Adrienne Lowe) — бывший шеф-повар из Атланты, сейчас разработчик на Python, делится рассказами, конспектами конференций и рецептами в блоге по кулинарии и программированию «Программирование с ножами» (http://codingwithknives.com). Она организует конференции PyLadiesATL и Django Girls Atlanta и ведет блог «Your Django Story» на сайте организации «Django Girls», где публикует интервью с женщинами, пишущими на Python. Адрианна работает инженером техподдержки в Emma Inc., директором по развитию в Django Software Foundation, а также входит в основную команду технических писателей. Она предпочитает рукописные письма электронной почте и с детства собирает коллекцию марок.

Монте Миланук (Monte Milanuk) сделал немало ценных замечаний.

Признательности и благодарности

Моему редактору. Редактором этого издания является **Дон Шанафельт** (Dawn Schanafelt), и эта книга стала гораздо, гораздо лучше благодаря участию Дон. Дон не просто выдающийся редактор, ее внимание к деталям и умение правильно излагать мысли значительно улучшили все, что было написано в этой книге. В издательстве *O'Reilly Media* вошло в привычку нанимать ярких, дружелюбных, способных людей, и Дон является олицетворением всех этих достоинств.

Команде O'Reilly Media. Нам потребовалось четыре года, чтобы подготовить это издание «Head First Python» (это долгая история). Вполне естественно, что многие из команды *O'Reilly Media* были вовлечены в эту работу. **Кортни Нэш** (Courtney Nash) уговорила меня «быстро переписать» книгу в 2012 году, а затем проект немного раздулся. Кортни была первым редактором этого издания и была рядом, когда произошла катастрофа и казалось, что книга обречена. Когда все постепенно вернулось на круги своя, Кортни отвлекли на большие и лучшие проекты O'Reilly Media. Передав бразды редактирования в 2014 году очень занятой **Меган Бланшетт** (Meghan Blanchette), которая наблюдала (я полагаю, с возрастающим ужасом), как задержка накладывалась на задержку, а работа над этой книгой выбивалась из графика раз за разом. Мы вернулись в нормальное русло, когда Меган отправилась на новые пастбища, и Дон взяла на себя работу в качестве редактора этой книги. Это было год назад, и большая часть книги — 12¾ глав — была написана под постоянным бдительным надзором Дон. Как я уже упоминал, O'Reilly Media нанимает хороших людей, и Кортни и Меган внесли большой вклад в редактирование и поддержку, и я им признателен. Особую благодарность заслужили **Морин Спенсер** (Maureen Spencer), **Хезер Шерер** (Heather Scherer), **Карен Шанер** (Karen Shaner) и **Крис Паппас** (Chris Pappas) за работу «за кулисами». Благодарим также невидимых, незаметных героев, известных как **отдел подготовки к печати**, которые приняли мои главы в *InDesign* и превратили их в этот готовый продукт. Они сделали большую работу.

Искреннее спасибо **Берту Бейтсу** (Bert Bates), который вместе с **Кэти Сьерра** (Kathy Sierra) создал эту серию книг с их замечательной «Head First Java» («Изучаем Java», Кэти Сьерра, Берт Бейтс, «Эксмо», 2012). Берт провел много времени, работая со мной и указывая верное направление для этого издания.

Друзьям и коллегам. Моя благодарность **Найджелу Вайту** (Nigel Whyte) (заведующему кафедрой вычислительной техники в Технологическом институте, Карлоу) за поддержку в работе. Многие мои студенты проходили большую часть этого материала во время учебы, надеюсь, они улыбнутся, когда увидят пару (или даже больше) примеров с наших занятий на страницах книги.

Еще раз спасибо **Дэвиду Гриффитсу** (David Griffiths) (моему «соучастнику» по книге «Head First Programming») за то, что сказал мне в один особенно сложный момент перестать беспокоиться обо всем подряд и просто *написать эту проклятую книгу*! Это был отличный совет. Здорово, что Дэвид вместе с Дон (его женой и соавтором «Head First») всегда доступны по электронной почте. Обязательно почитайте замечательные книги из серии «Head First» Дэвида и Дон.

Семье. Моя семья (жена **Дейдра (Deirdre)**, дети **Иосиф (Joseph)**, **Аарон** (Aaron) и **Айдин (Aideen)**) выдержала четыре года подъемов и падений, остановок и начинаний, вздохов и фырканий и получила опыт, изменяющий жизнь, через который мы все сумели пройти, к счастью, все еще невредимыми. Эта книга выжила, я выжил, наша семья выжила. Я очень благодарен им и люблю их всех, и я знаю, что не нужно говорить об этом, но все-таки скажу: «Я делаю это для вас».

Другим людям. Команда технических редакторов проделала отличную работу: смотрите их мини-резюме на предыдущей странице. Я рассмотрел все их рекомендации, исправил все ошибки, которые они нашли, и всегда был очень горд, когда кто-нибудь из них находил время, чтобы сказать мне, что я проделал большую работу. Я очень благодарен им всем.

1 Основы

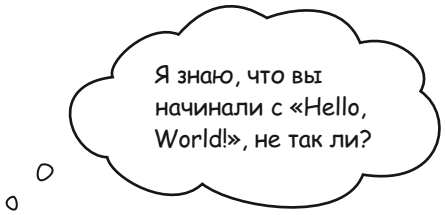
Начнем поскорее

Начнем программировать на Python как можно скорее.

В этой главе мы ознакомимся с основами программирования на Python и сделаем это в характерном для нас стиле: с места в карьер. Через несколько страниц вы запустите свою первую программу. К концу главы вы сможете не только запускать типичные программы, но также понимать их код (и это еще не все!). Попутно вы познакомитесь с некоторыми особенными чертами языка **Python**. Итак, не будем больше тратить время. Переверните страницу — и вперед!

Ломаем традиции

Возьмите почти любую книгу по программированию, и первое, что вы увидите, будет пример *«Hello World»*.



Но нет, мы не такие

Это книга из серии Head First, и мы все делаем не так, как вы привыкли. Другие книги по традиции начинаются с того, что вам показывают, как написать программу «Hello World» на рассматриваемом языке программирования. Однако в случае с Python вы используете одно выражение, вызывающее встроенную функцию `print`, которая, в свою очередь, выводит сообщение «Hello, World!» на экран. Не слишком интересно... да и чему вы тут научитесь?

Поэтому мы не собираемся показывать вам программу «Hello World» на Python, поскольку так вы ничего не узнаете. Мы пойдем другим путем...

Начнем с более содержательного примера

Эту главу мы начнем с примера, который несколько больше и, следовательно, полезнее, чем «Hello World».

Забегая наперед, мы скажем, что наш пример немного *надуманный*: он делает что-то, но в дальнейшем не будет вам особо полезен. И все же мы выбрали его, чтобы предоставить «транспортное средство», с помощью которого можно будет покрыть большую часть Python в максимально короткий отрезок времени, насколько это возможно. И мы обещаем, что к тому времени как вы закончите работать с первым примером программы, вы будете знать достаточно, чтобы написать «Hello World» на Python без нашей помощи.

Запрыгиваем

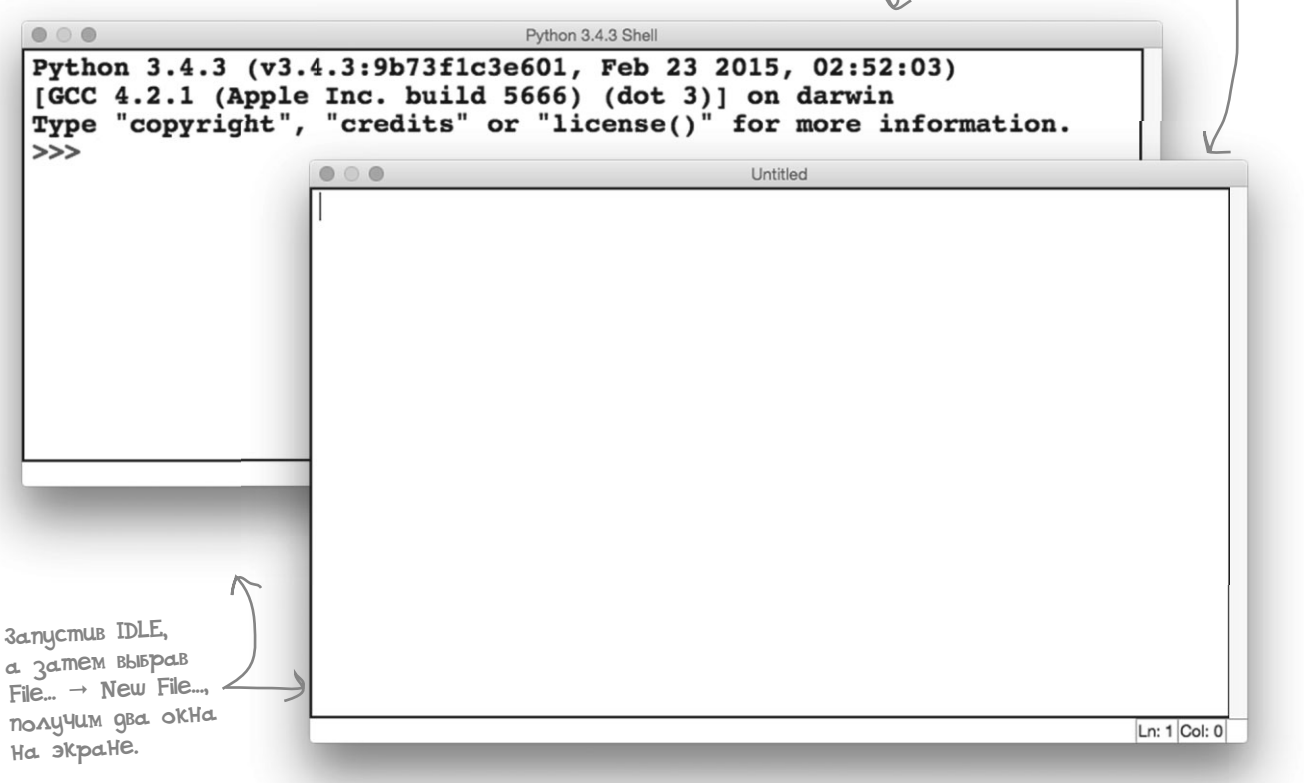
Если у вас на компьютере еще не установлена версия Python 3, сделайте паузу и перейдите к Приложению А, где вы найдете пошаговые инструкции по его установке (это займет всего несколько минут, обещаю).

Установив последний выпуск Python 3, вы будете готовы начать программировать на Python, а для простоты — пока — мы собираемся использовать встроенную в Python интегрированную среду разработки (IDE).

IDLE Python — это все, что нужно для начала

Установив Python 3, вы получите в комплекте очень простую, но полезную IDE под названием IDLE. Хотя есть много различных способов запустить код на Python (и вы встретите многие из них в этой книге), IDLE — это все, что нужно для начала.

Запустите IDLE на вашем компьютере, а затем последовательно выберите пункты меню *File...* →*New File...*, чтобы открыть новое окно редактирования. Когда вы выполните это на своем компьютере, должны появиться два окна: одно с заголовком Python Shell, а другое — «Untitled»:



Назначение окон IDLE

Оба окна IDLE важны.

Первое окно, командная оболочка Python, — это REPL-среда, используемая для запуска фрагментов кода Python, обычно одной инструкции за раз. Чем больше вы будете работать с Python, тем больше вам будет нравиться командная оболочка Python, и вы будете часто использовать ее по мере чтения этой книги. Однако сейчас нас больше интересует второе окно.

Второе окно, Untitled, — это окно текстового редактора, в котором можно написать законченную программу на Python. Это не самый лучший редактор в мире (этой чести удостоился *<вставьте сюда название своего любимого редактора>*), но редактор IDLE достаточно удобен и обладает большим набором современных возможностей, уже встроенных в него, включая подсветку синтаксиса и тому подобное.

Для умников

Что значит REPL?

Это профессиональное жаргонное сокращение от «read-eval-print-loop» (прочитать-выполнить-напечатать-повторить) и означает интерактивный программный инструмент, который позволяет экспериментировать с какими только пожелаете кусочками кода. Чтобы узнать больше, чем вам нужно знать, пройдите по ссылке https://ru.wikipedia.org/wiki/REPL.

Поскольку наш принцип — «с места в карьер», давайте двигаться вперед и введем небольшую программу на Python в это окно. Когда вы закончите вводить код, приведенный ниже, используйте пункты меню *File...* → *Save...*, чтобы сохранить программу под названием `odd.py`.

Убедитесь, что введенный код *в точности* соответствует приведенному здесь:

Не беспокойтесь сейчас о том, что этот код делает. Просто введите его в окно редактирования. Убедитесь, что вы сохранили его как «odd.py», прежде чем продолжить.

odd.py - /Users/Paul/Desktop/_NewBook/ch01/odd.py (3.4.3)

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

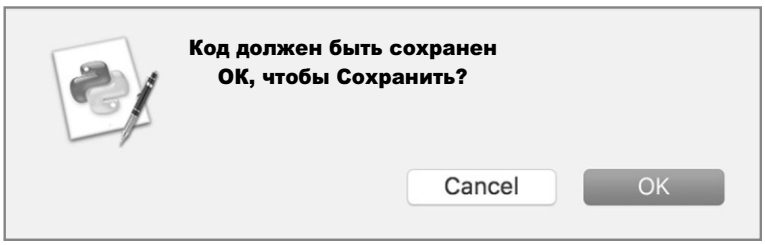
Ln: 15 Col: 0

Итак... *что теперь?* Готов поспорить, что вам не терпится запустить этот код, не так ли? Давайте сделаем это. В окне редактирования с кодом (которое показано выше) нажмите клавишу F5 на клавиатуре. Это может вызвать несколько разных событий...

Что будет дальше...

Если код выполнится без ошибок, переверните страницу и продолжайте чтение.

Если вы забыли сохранить свой код, *прежде* чем попытались запустить его, IDLE пожалуется, так как вы должны *в первую очередь* сохранить любой новый код в файл. Если вы не сохранили код, то увидите сообщение, подобное этому:

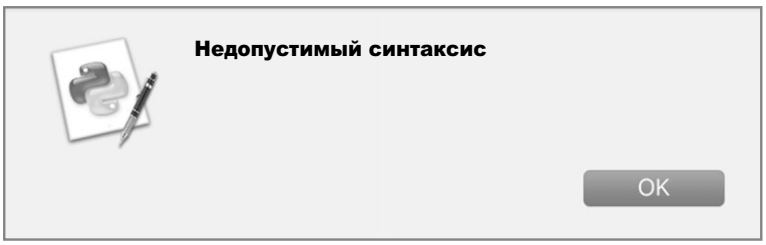


По умолчанию, IDLE не будет выполнять код, который не сохранен.

Щелкните на кнопке OK, после чего задайте имя для вашего файла. Мы выбрали для файла имя `odd` и добавили расширение `.py` (это соглашение Python, которого стоит придерживаться):

Вы можете использовать любое имя для вашей программы, но, вероятно, будет лучше, если вы воспользуетесь тем же именем, что и мы.

Если теперь (после сохранения) код выполнился без ошибок, переверните страницу и *продолжайте читать*. Однако если где-то в коде вы допустили синтаксическую ошибку, то увидите такое сообщение:



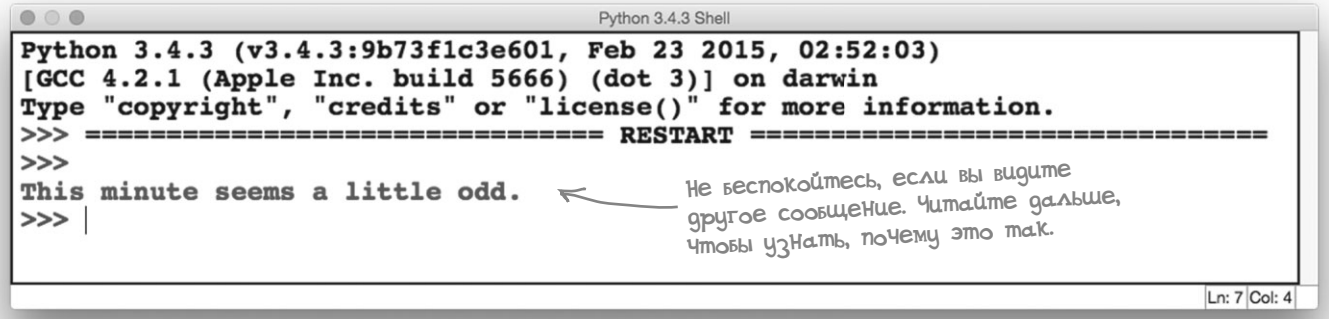
Можно без сомнения сказать, что IDLE не слишком хороша в поиске синтаксических ошибок. Но нажмите кнопку OK, и большой красный блок укажет на то место, где, по мнению IDLE, есть проблема.

Щелкните на кнопке OK, а затем обратите внимание, где именно, по мнению IDLE, содержится синтаксическая ошибка: посмотрите на большой красный блок в окне редактирования. Убедитесь, что ваш код в точности соответствует нашему, сохраните файл еще раз, а затем нажмите клавишу F5, чтобы IDLE выполнила код еще раз.

Нажмите F5 для запуска вашего кода

Нажатие клавиши F5 выполнит код в окне редактирования IDLE, активном в данный момент, если он не содержит ошибок времени выполнения. Если в коде встретится ошибка времени выполнения, вы увидите сообщение об ошибке (красным цветом) и **стек вызовов**. Прочитайте сообщение, а затем вернитесь к окну редактирования, чтобы убедиться, что введенный вами код в точности такой же, как у нас. Сохраните ваш измененный код, а затем снова нажмите клавишу F5. Когда мы нажимаем F5, активизируется окно с командной оболочкой Python, и вот что мы увидим:

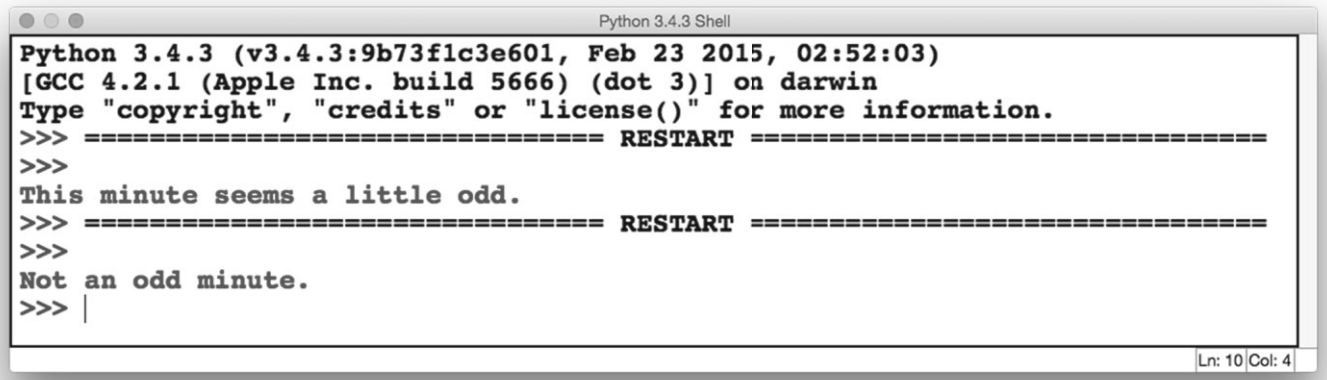
С этого момента «окно IDLE для редактирования текста» мы будем называть просто «окно редактирования».



Не беспокойтесь, если вы видите другое сообщение. Читайте дальше, чтобы узнать, почему это так.

В зависимости от текущего времени, вы можете увидеть сообщение *Not an odd minute* («Нечетное число минут»). Не беспокойтесь, если так получилось, потому что эта программа выводит на экран то одно, то другое сообщение, в зависимости от четности или нечетности значения минут текущего времени на вашем компьютере (мы уже говорили, что этот пример *надуманный*, не так ли?). Если вы подождете минуту, щелкнете на окне редактирования, чтобы выбрать его, а затем нажмете клавишу F5 еще раз, ваш код запустится снова. На этот раз вы увидите другое сообщение (подразумевается, что вы выждали требуемую минуту). Не стесняйтесь, запускайте этот код так часто, как хотите. Вот то, что мы видели, когда мы (очень терпеливо) ждали требуемую минуту:

Нажатие клавиши F5 в окне редактирования запустит ваш код, затем в окне с командной оболочкой Python появится результат.

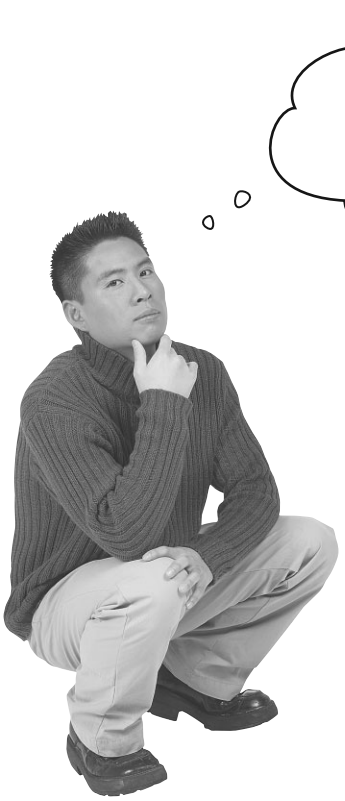


Давайте потратим немного времени и разберемся, как выполняется этот код.

Код запускается незамедлительно

Когда IDLE просит Python выполнить код в окне редактирования, Python немедленно начинает выполнение кода с первой строки в файле.

Тем из вас, кто переходит на Python с одного из C-подобных языков, следует обратить внимание, что в Python не существует понятия функции или метода `main()`. Тут также нет понятия процесса, известного как «редактирование-компиляция-компоновка-запуск». В Python вы редактируете свой код, сохраняете его и запускаете *незамедлительно*.



Хороший вопрос, в котором легко запутаться.

Запомните: «Python» — это название языка программирования, а «IDLE» — это название встроенной IDE для Python.

При установке Python 3 на компьютер также устанавливается **интерпретатор**. Это механизм, который выполняет код на Python. Интерпретатор тоже известен под названием «Python», что увеличивает путаницу. По-хорошему, каждый, ссылаясь на этот механизм, должен использовать более правильное название «интерпретатор Python». Но, увы, никто этого не делает.

С этого момента мы будем использовать слово «Python» для обозначения языка, а слово «интерпретатор» — для обозначения механизма, выполняющего код на Python. Слово «IDLE» относится к IDE, которая принимает ваш код на Python и запускает его с помощью интерпретатора. Фактически всю работу делает интерпретатор.

Это не Глупые вопросы

В: Интерпретатор Python — это что-то вроде Java VM?

О: И да и нет. Да, потому что интерпретатор выполняет ваш код. Нет относится к тому, как он это делает. В Python нет понятия исходного кода, который компилируется в «выполняемый файл». В отличие от Java VM, интерпретатор не запускает .class файлы, он просто запускает ваш код.

В: Но, конечно же, компиляция должна произойти на каком-то этапе?

О: Да, это происходит, но интерпретатор не раскрывает этот процесс для программиста на Python (вас). Эти детали вас не касаются. Все, что вы видите, — это код, который работает, поскольку IDLE делает всю сложную работу, взаимодействуя с интерпретатором от вашего имени. Мы расскажем больше об этом процессе далее по ходу книги.

Выполнение кода, одна инструкция за раз

Ниже еще раз приводится программа со страницы 4:

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

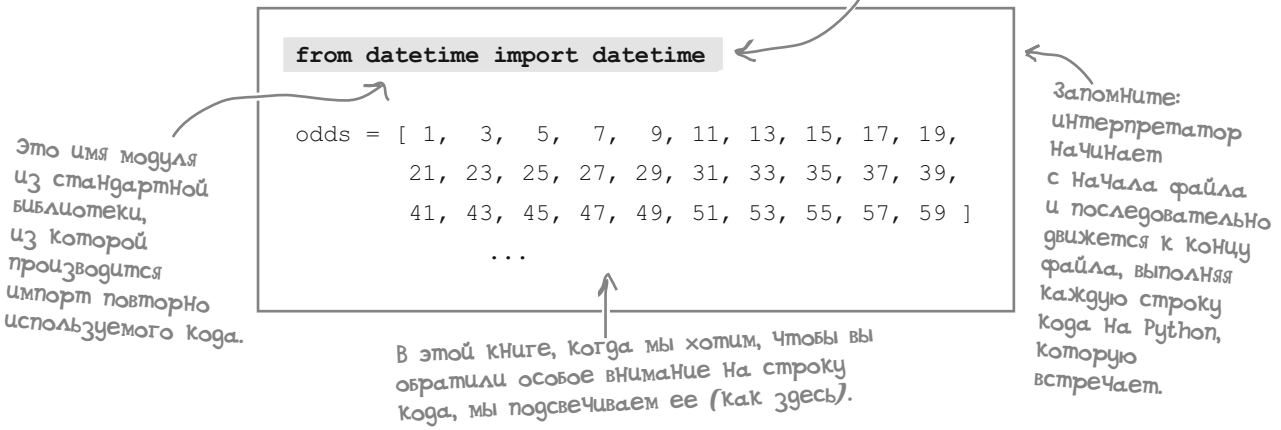
Думайте о модулях как о коллекциях связанных функций.

Давайте побудем интерпретатором Python

Давайте потратим немного времени и выполним этот код так, как это делает интерпретатор, строку за строкой, от *начала* файла до *конца*.

Первая строка **импортирует** некоторую встроенную функциональность из **стандартной библиотеки** Python, которая содержит большой запас программных модулей, предоставляющих множество заранее подготовленного (и высококачественного) повторно используемого кода.

В нашей программе, в частности, мы запросили один из субмодулей модуля `datetime` стандартной библиотеки. Тот факт, что субмодуль также называется `datetime`, сбивает с толку, но так уж это работает. Субмодуль `datetime` реализует механизм для работы с временем, как вы увидите на следующих нескольких страницах.



Функции + модули = стандартная библиотека

Стандартная библиотека Python очень *богата* и предоставляет много кода для повторного использования.

Давайте посмотрим на другой модуль, под названием `os`, который предоставляет независимый от платформы способ взаимодействия с операционной системой (скоро мы вернемся к модулю `datetime`). Рассмотрим только одну из его функций: `getcwd`, которая при вызове возвращает *текущий рабочий каталог.*

Вот типичный пример *импорта* и *вызова* этой функции в программе на Python:

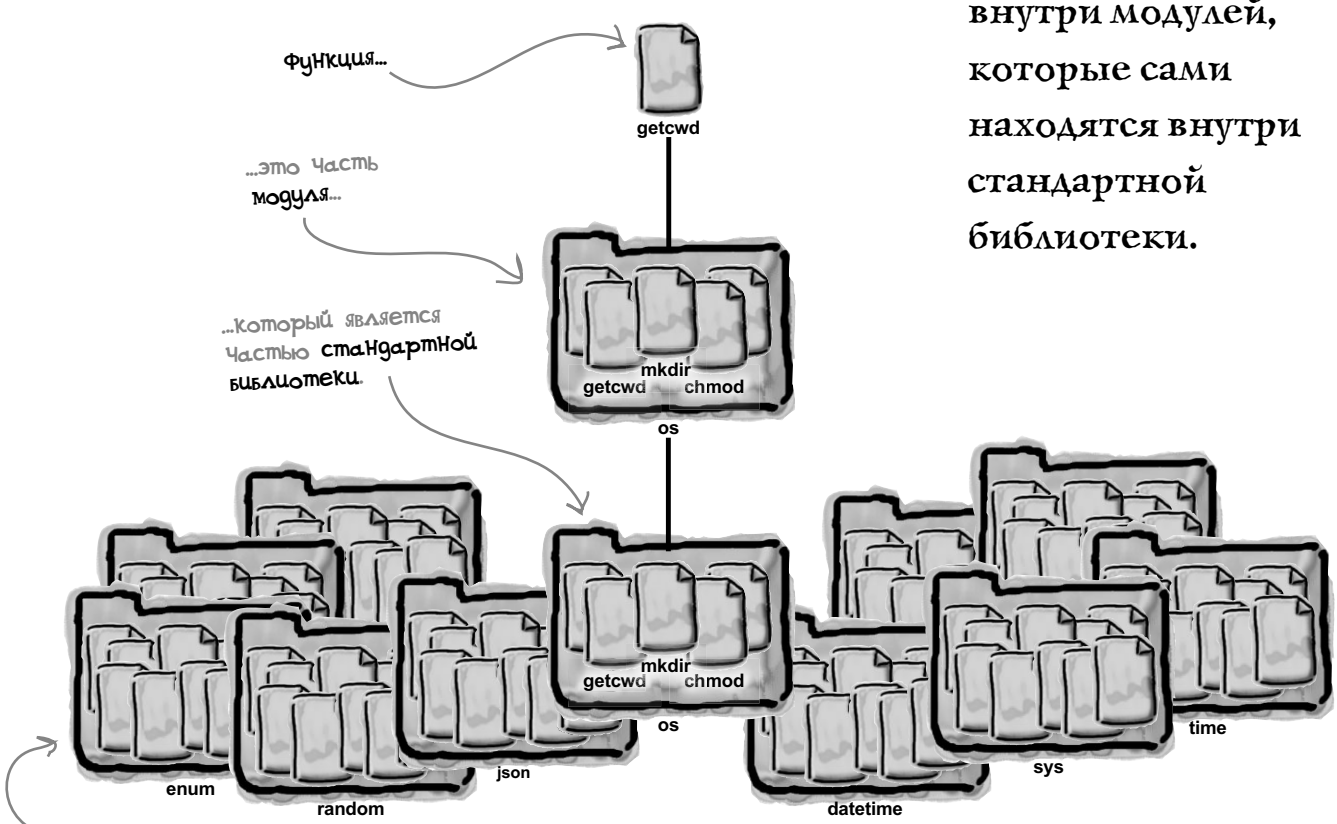
Импортируем функцию из модуля...

```
from os import getcwd

where_am_I = getcwd()
```

...затем вызываем, когда потребуется.

Набор связанных функций составляет программный модуль, и в стандартной библиотеке есть *много* модулей:

Функции находятся внутри модулей, которые сами находятся внутри стандартной библиотеки.



Для нас пока не важно, что делает каждый из этих модулей. Некоторые из них вкратце описываются на следующих страницах, но далее в книге мы будем рассматривать их более подробно.

Стандартная библиотека под лупой

Стандартная библиотека — это бриллиант в короне Python, она содержит модули на все случаи жизни, которые помогут, например, упаковать или распаковать ZIP-архив, отправить или принять электронную почту, обработать HTML-страницу. Стандартная библиотека даже включает веб-сервер, а также механизм для работы с популярными базами данных *SQLite*. В этом *коротком обзоре* мы познакомим вас с некоторыми часто используемыми модулями стандартной библиотеки. Следуя за нами, вы можете вводить представленные примеры, как показано, в командную строку >>> (в IDLE). Если вы сейчас видите окно редактирования IDLE, выберите пункт меню *Run... →Python Shell*, чтобы перейти к командной строке >>>.

Давайте исследуем систему, в которой работает интерпретатор. Хотя Python гордится своей кросс-платформенностью, то есть код, написанный на одной платформе, может выполняться (как правило, без изменения) на другой, иногда важно знать, что вы работаете, скажем, в *Mac OS X*. Больше узнать о системе вам поможет модуль `sys`. Чтобы определить операционную систему, сначала импортируйте модуль `sys`, а затем обратитесь к атрибуту `platform`:

```
>>> import sys
>>> sys.platform
'darwin'
```

Поэкспериментируем в darwin (ядро Mac OS X).
Импортируем нужный модуль и обращаемся к атрибуту.

Модуль `sys` — это хороший пример повторно используемого модуля, который в первую очередь предоставляет доступ к предустановленным атрибутам (таким как `platform`). В следующем примере мы определим, какая версия Python используется, и передадим это значение функции `print` для вывода на экран:

```
>>> print(sys.version)
3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
```

Информация об используемой версии Python.

Модуль `os` — это хороший пример повторно используемого модуля, который в первую очередь предоставляет доступ к богатому набору функций и реализует универсальный способ взаимодействия кода на Python с операционной системой, независимо от ее вида.

Ниже показано, как получить имя папки, в контексте которой выполняется код, используя функцию `getcwd`. Перед вызовом функции импортируем модуль:

```
>>> import os
>>> os.getcwd()
'/Users/HeadFirst/CodeExamples'
```

Импортируем модуль, затем вызываем нужную функцию.

Вы можете получить доступ к системным переменным окружения, как ко всем сразу (используя атрибут `environ`) так и индивидуально (используя функцию `getenv`):

```
>>> os.environ
'environ({'XPC_FLAGS': '0x0', 'HOME': '/Users/HeadFirst', 'TMPDIR': '/var/
 folders/18/t93gmhc546b7b2cngfhz10100000gn/T/', ... 'PYTHONPATH': '/Applications/
 Python 3.4/IDLE.app/Contents/Resources', ... 'SHELL': '/bin/bash', 'USER':
'HeadFirst'})'
>>> os.getenv('HOME')
'/Users/HeadFirst'
```

Атрибут «environ» содержит множество данных.

С помощью «getenv» можно получить конкретный атрибут (из данных, содержащихся в «environ»).

Стандартная библиотека под лупой, продолжение

Работать с датами (и временем) приходится много, и стандартная библиотека предоставляет модуль `datetime` чтобы помочь вам, когда вы работаете с этим типом данных. Функция `date.today` возвращает текущую дату:

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2015, 5, 31)
```

Текущая дата.

Странный способ отображения текущей даты, не находите? Отдельно число, месяц и год можно получить, добавляя соответствующий атрибут доступа к вызову date.today:

```
>>> datetime.date.today().day
31
>>> datetime.date.today().month
5
>>> datetime.date.today().year
2015
```

Части, составляющие текущую дату.

Также можно вызвать функцию `date.isoformat` и передать ей текущую дату, чтобы получить более удобочитаемую версию текущей даты, которая получается преобразованием даты в строку с помощью `isoformat`:

```
>>> datetime.date.isoformat(datetime.date.today())
'2015-05-31'
```

Текущая дата в виде строки.

Теперь перейдем к времени, которого, кажется, всем нам не хватает. Может ли стандартная библиотека сказать, который сейчас час? Да. Для этого, после импортирования модуля `time`, надо вызвать функцию `strftime`, определив формат отображения времени. Например, если вам интересно узнать текущее время в часах (`%I`) и минутах (`%M`) в 12-часовом формате:

```
>>> import time
>>> time.strftime("%I:%M")
'11:55'
```

Боже мой! Это время?

А как узнать день недели и определить текущую половину суток — до полудня или после? В этом вам помогут спецификаторы `%A%p`:

```
>>> time.strftime("%A %p")
'Sunday PM'
```

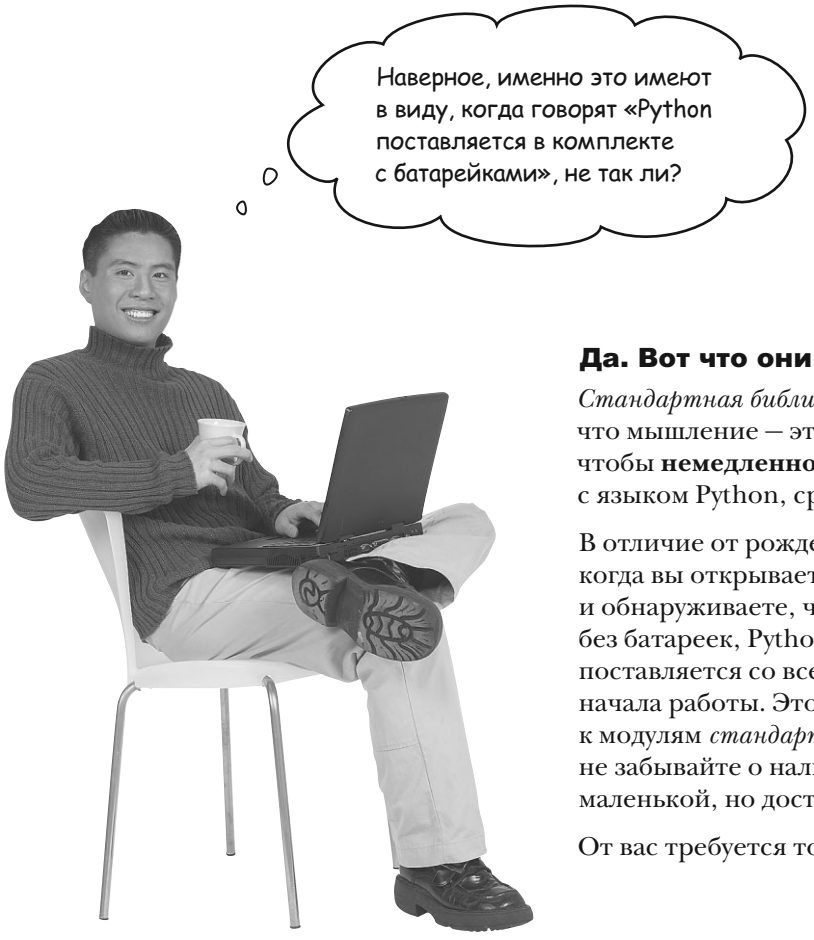
Мы выяснили, что сейчас воскресенье, без пяти минут полночь... возможно, пора спать?

В качестве примера функциональных возможностей *стандартной библиотеки* представьте, что у вас есть некоторые HTML документы, и вы беспокоитесь, что они могут содержать потенциально опасные теги `<script>`. Вместо того чтобы просматривать разметку HTML и удалять теги, почему бы не экранировать проблемные угловые скобки с помощью функции `escape` из модуля `html`? Если у вас есть экранированный HTML-документ и вы хотите вернуть его в оригинальное представление, то функция `unescape` сделает это. Пример использования обеих функций:

```
>>> import html
>>> html.escape("This HTML fragment contains a <script>script</script> tag.")
'This HTML fragment contains a &lt;script&gt;script&lt;/script&gt; tag.'
>>> html.unescape("I &hearts; Python's &lt;standard library&gt;.")
"I ♥ Python's <standard library>."
```

Преобразование разметки HTML в экранированный текст и обратно.

Батарейки в комплекте



Да. Вот что они имеют в виду.

Стандартная библиотека настолько богата, что мышление — это все, что вам нужно, чтобы **немедленно приступить к работе** с языком Python, сразу после его установки.

В отличие от рождественского утра, когда вы открываете новую игрушку и обнаруживаете, что она поставляется без батареек, Python вас не разочарует; он поставляется со всем необходимым для начала работы. Это относится не только к модулям *стандартной библиотеки*: не забывайте о наличии в комплекте IDLE — маленькой, но достаточно удобной IDE.

От вас требуется только начать писать код.

Это не Глупые вопросы

В: Как узнать, что именно делает конкретный модуль стандартной библиотеки?

О: Документация Python содержит все ответы на вопросы о стандартной библиотеке. Вот отправная точка: *https://docs. python.org/3/library/index.html*.

Для умников

Стандартная библиотека — это не единственное место, где можно найти отличные модули для использования в своих программах. Сообщество Python поддерживает обширную коллекцию сторонних модулей, некоторые из них рассмотрены далее. Если вы хотите предварительно с ними ознакомиться, зайдите в репозиторий сообщества по адресу http://pypi.python.org.

Встроенные структуры данных

Помимо первосортной *стандартной библиотеки* Python имеет также мощные встроенные **структуры данных**. Одной из них является **список**, который может рассматриваться как очень мощный *массив*. Подобно массивам во многих других языках программирования, списки в Python заключаются в квадратные скобки (`[]`).

Подобно массивам, списки могут хранить данные любых типов.

Следующие три строки в нашей программе (как показано ниже) присваивают *литеральный* список нечетных чисел переменной с именем `odds`. В этом коде `odds` — это *список целых чисел*, но списки в Python могут хранить *любые* данные *любых* типов. Более того, вы можете даже смешивать в списке данные разных типов (если это необходимо). Обратите внимание, что список `odds` располагается в трех строках, несмотря на то что это одно выражение. Это нормально, поскольку интерпретатор не будет считать выражение законченным, пока не найдет закрывающую скобку (`]`), которая соответствует открывающей (`[`). Как правило, **в Python конец строки отмечает конец инструкции**, но могут быть исключения из общего правила, и многострочный список — один из них (далее нам встретятся и другие).

Это новая переменная с именем «odds», которой присваивается список нечетных чисел.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
               ...
```

Список нечетных чисел, заключенный в квадратные скобки. Это одна инструкция, которая занимает три строки кода, и это нормально.

Со списками можно сделать многое, но мы отложим их дальнейшее обсуждение до следующей главы. Все, что вам нужно знать сейчас: этот список *существует* в настоящее время, он был *присвоен* переменной `odds` (благодаря использованию **оператора присваивания**, =) и *содержит* показанные числа.

Python умеет работать со всеми обычными операторами, включая <, >, <=, >=, ==, !=, а также с оператором =, оператором присваивания.

Значения переменным в Python присваются динамически

Прежде чем перейти к следующей строке кода, возможно, стоит сказать несколько слов о переменных, особенно для тех, кто привык объявлять переменные и их типы *перед* использованием (как это характерно для статически типизированных языков программирования).

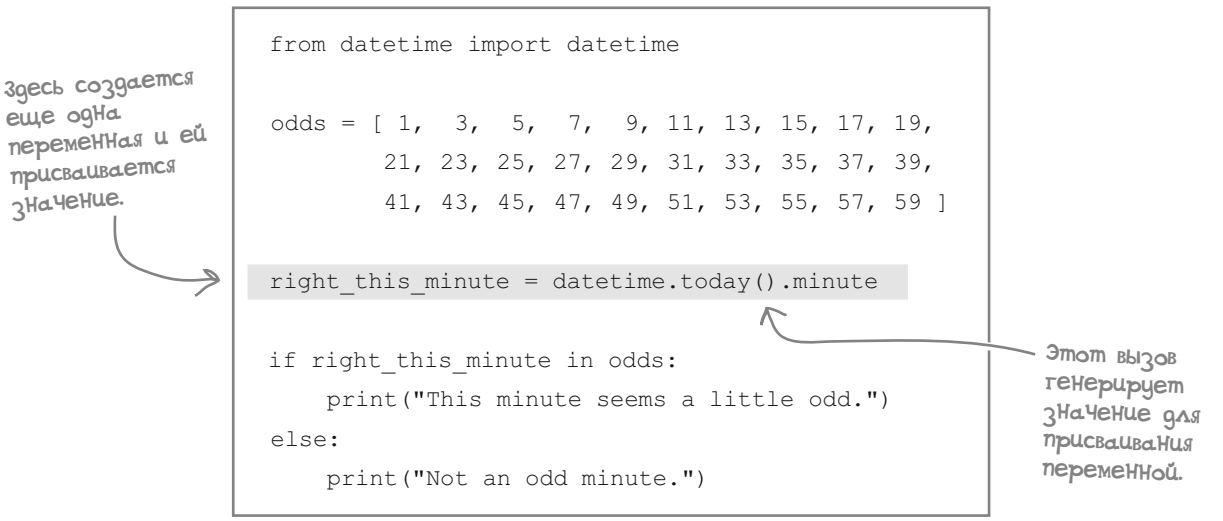
В Python переменные начинают существовать при первом их использовании, и **их типы не требуют предварительного определения**. Переменные в Python принимают те же типы, что имеют объекты, которые им присваиваются. В нашей программе переменной `odds` присваивается список чисел, поэтому `odds` в этом случае становится списком.

Давайте посмотрим на другую инструкцию присваивания переменной. К счастью, она находится в следующей строке нашей программы.

Вызов метода возвращает результат

Третья строка в нашей программе — еще одна **инструкция присваивания.**

В отличие от предыдущей, она присваивает другой, новой переменной с именем `right_this_minute` не структуру данных, а **результат** вызова метода. Взгляните еще раз на третью строку кода:

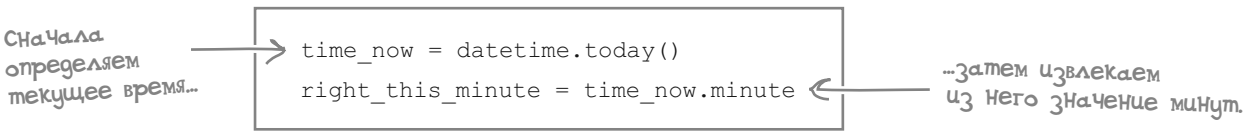


Вызов функции из модуля

Третья строка вызывает метод с именем `today`, входящий в субмодуль `datetime`, который *сам* является частью модуля `datetime` (мы уже отмечали некоторую запутанность этой стратегии именования). Говорить о вызове метода позволяют стандартные постфиксные круглые скобки: `()`.

Вызов `today` возвращает «объект времени», содержащий информацию о текущем времени, раздробленную на части. Это обычные **атрибуты** текущего времени, обратиться к которым можно с помощью синтаксиса **точечной нотации** (записи через точку). В этой программе нас интересует атрибут минут, обратиться к которому можно, добавив `.minute` к вызову метода, как показано выше. Полученное значение присваивается переменной `right_this_minute`. Вы можете думать об этой строчке кода, рассуждая таким образом: *создается объект, представляющий текущее время, из него извлекается значение атрибута minute, которое затем присваивается переменной.* Заманчиво *разделить* одну строчку кода на две, чтобы сделать ее «проще для понимания», как показано ниже:

Синтаксис с точечной нотацией часто будет встречаться далее в книге.



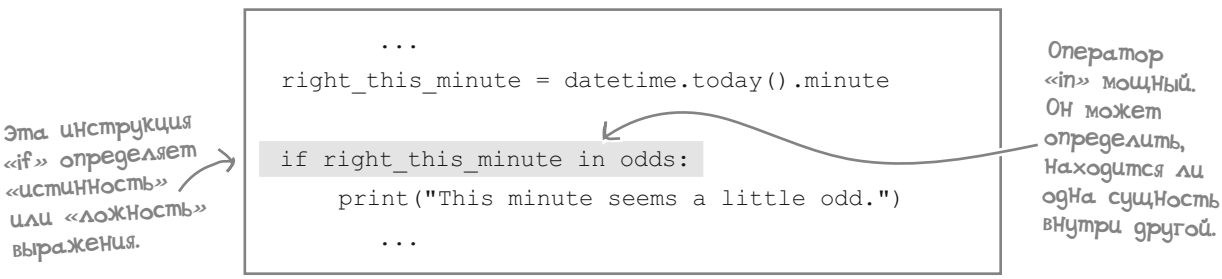
Вы можете сделать это (если хотите), но большинство программистов на Python предпочитают **не** создавать новые переменные (в этом примере это `time_now`), *если* они не потребуются где-нибудь дальше в программе.

Принятие решения о запуске блока кода

На этом этапе у нас есть список чисел `odds`. У нас также есть значение минут `right_this_minute`. Чтобы проверить, является ли значение минут в `right_this_minute` нечетным числом, нам надо каким-то образом определить, содержится ли оно в списке нечетных чисел. Но как это сделать?

Оказывается, в Python подобные проверки делаются очень просто. Помимо обычных операторов сравнения, которые можно встретить в любом языке программирования (таких как >, <, >=, <= и так далее), в Python имеется несколько собственных «супероператоров», один из которых — `in`.

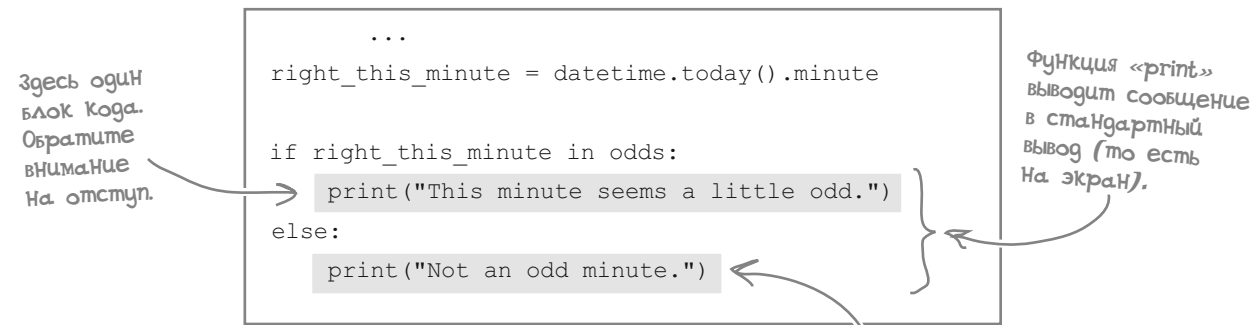
Оператор `in` проверяет, находится ли что-то одно *внутри* другого. Посмотрите на следующую строчку в программе, которая использует оператор `in` для проверки, не находится ли `right_this_ minute` *в* списке `odds`:



Оператор `in` возвращает `True` или `False`. Как вы уже догадались, если значение `right_this_minute` содержится в `odds`, выражение в инструкции `if` вернет `True`, и она выполнит ассоциированный с ней блок кода.

Блоки в Python легко заметны, поскольку они всегда выделены отступом.

В нашей программе есть два блока, каждый из них содержит по одному вызову функции `print`. Эта функция может отображать сообщения на экране (дальше мы увидим много примеров ее применения). Когда вы вводили код этой программы в окно редактирования, вы могли заметить, что IDLE помогает выдерживать отступы автоматически. Это очень полезно, но не забывайте проверять, чтобы отступы, расставленные IDLE, соответствовали тому, что вам нужно:



Вы заметили, что здесь нет фигурных скобок?

Что случилось с моими фигурными скобками?

Это не значит, что в Python не применяются фигурные скобки. Они используются, но — как мы увидим в главе 3 — чаще для обозначения границ данных, чем фрагментов кода.

Блоки кода в любой программе на Python легко заметны по отступам. Отступы помогают вашему мозгу быстро идентифицировать блоки в процессе чтения кода. Другим визуальным ключом для поиска является символ двоеточия (`:`), который используется для введения блока кода, ассоциированного с любой управляющей инструкцией Python (такой как `if`, `else`, `for` и подобными). Вы увидите множество примеров такого использования двоеточия по мере чтения книги.

Оформляйте отступы единообразно: или символами табуляции, или пробелами. Не смешивайте их, иначе интерпретатор может отказаться выполнять ваш код!

Двоеточие вводит блок кода

Двоеточие (`:`) важно, поскольку вводит новый блок кода, который должен быть смещен на один отступ вправо. Если вы забыли сделать отступ для своего кода после двоеточия, интерпретатор выдаст ошибку.

Не только инструкция `if` в нашем примере содержит двоеточие, его также имеет `else`. Вот весь код целиком еще раз:

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Двоеточия вводят блоки с дополнительными отступами.

Мы почти закончили. Осталось обсудить последнюю инструкцию.

Какие варианты может иметь «if»?

Мы почти закончили с примером нашей программы, осталась только одна строка, которую нужно обсудить. Не очень длинная, но очень важная: инструкция `else` идентифицирует блок кода, который выполняется, когда условное выражение в инструкции `if` возвращает значение `False`.

Рассмотрим подробнее инструкцию `else` в нашем примере: мы убрали отступ перед ней для выравнивания с соответствующей инструкцией `if`:

```
if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Видите двоеточие?

Возможно, вы заметили, что «else» используется без отступа для выравнивания с «if»?

Я думаю, что если есть «else», должно быть и «else if», или в Python это пишется как «elseif»?

Новички, начинающие осваивать Python, часто забывают ставить двоеточие.

Ни то ни другое. В Python это будет `elif`.

Если у вас есть несколько условий, которые нужно проверить в одной инструкции `if`, Python предоставляет для этого `elif` наравне с `else`. Вы можете использовать столько инструкций `elif` (каждую со своим собственным блоком кода), сколько понадобится.

Вот небольшой пример, который предполагает, что переменной с именем `today` ранее было присвоено строковое значение, представляющее сегодняшний день недели:

```
if today == 'Saturday':
    print('Party!!')
elif today == 'Sunday':
    print('Recover.')
else:
    print('Work, work, work.')
```

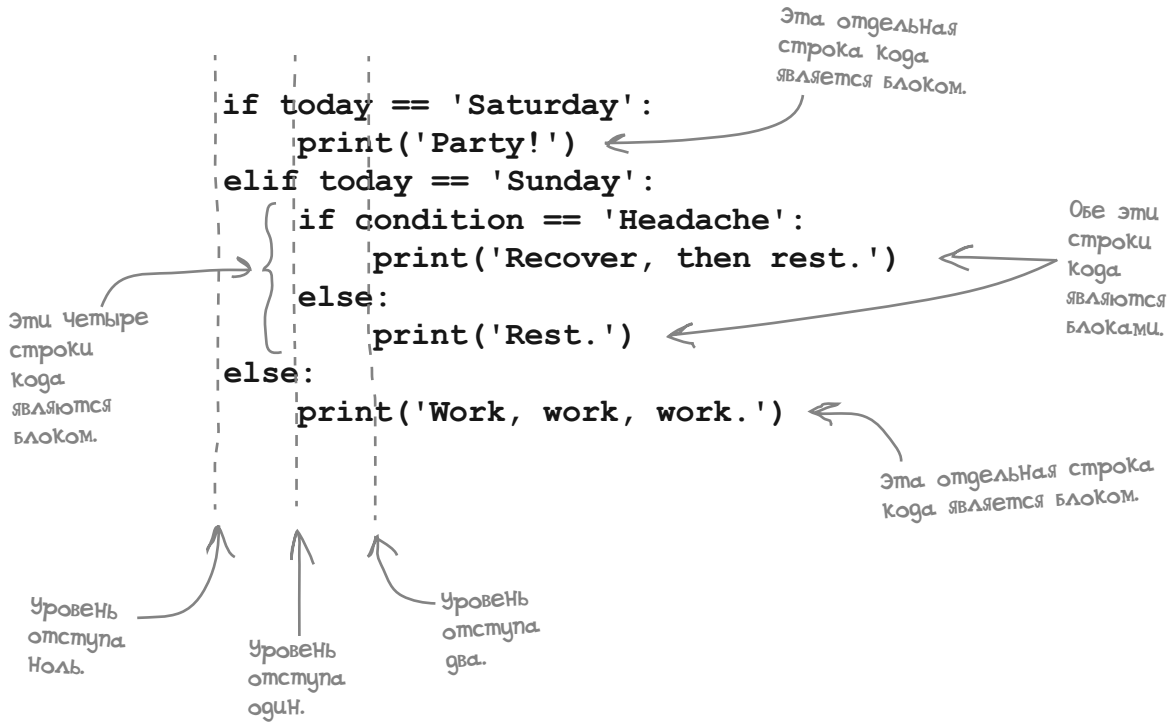
Три отдельных блока кода: один для «if», другой для «elif» и последний для «else», выполняющийся во всех остальных случаях.

Блоки кода могут содержать встроенные блоки

Каждый блок кода может содержать любое количество вложенных блоков, которые также отделяются отступами. Когда программисты на Python говорят о вложенных блоках, они имеют тенденцию говорить об **уровнях отступов**.

Начальный уровень отступов в программе, как правило, называют *первым* уровнем или (как принято начинать нумерацию во многих языках программирования) уровнем отступа *ноль*. Последующие уровни называют вторым, третьим, четвертым и так далее (или уровнем один, уровнем два, уровнем три и так далее).

Вот вариант кода примера с предыдущей страницы. Обратите внимание, что в инструкции `if` появились вложенные инструкции `if/else`, которые выполняются, когда `today` имеет значение `'Sunday'`. Предполагается также, что существует другая переменная, с именем `condition`, и ее значение отражает ваше текущее самочувствие. Мы указали, где находится каждый блок, а также на каком уровне отступов они находятся.



Важно заметить: код на одном уровне отступа связан с другим кодом на том же уровне отступа, только если все эти части кода находятся *в пределах одного блока*. В противном случае они принадлежат разным блокам, и тот факт, что они имеют один и тот же уровень отступа, не имеет значения. Важно помнить, что в Python отступы используются для отделения блоков кода.

Что мы уже знаем

Давайте повременим с рассмотрением нескольких последних строк кода и сделаем обзор того, что программа `odd.py` рассказала нам о Python.

КОНТРОЛЬНЫЙ СПИСОК

- Python поставляется со встроенной IDE, называемой IDLE, которая позволяет создавать, редактировать и запускать код на Python. Вам остается только ввести код, сохранить его и затем нажать F5.
- IDLE взаимодействует с интерпретатором Python, который автоматически выполняет процесс компиляции-сборки-запуска. Это позволяет вам сконцентрироваться на написании своего кода.
- Интерпретатор выполняет код (сохраненный в файле) сверху вниз, по одной строке за раз. В Python нет такого понятия, как функция/метод `main()`.
- Python поставляется с мощной стандартной библиотекой, включающей множество модулей (из которых `datetime` — лишь один пример).
- Также есть коллекция стандартных структур данных для использования в программах на Python. Одной из них является список, имеющий много общего с массивом.
- Переменные не требуется объявлять заранее. Когда вы присваиваете значение переменной, она динамически принимает тип присваиваемых данных.
- Вы можете принимать решение с помощью инструкций `if`/`elif`/`else`. Ключевые слова `if`, `elif`, `else` предваряют блоки кода, которые известны в мире Python как «наборы» (suite).
- Блоки кода легко различимы по отступам. Отступы — это единственный механизм группировки кода, который представляет Python.
- В дополнение к отступам блоки кода также предваряются двоеточием (`:`). Это синтаксическое требование языка.]

Давайте расширим программу, чтобы сделать больше.

И правда, чтобы описать работу этой короткой программы, нам требуется больше строк, чем содержит ее код. В этом заключается великая сила Python: *вы можете добиться многого всего несколькими строками кода.*

Просмотрите список выше еще раз, а затем переверните страницу и наблюдайте, как будет расширяться наша программа.

Расширим программу, чтобы сделать больше

Давайте расширим нашу программу для более глубокого изучения Python.

Сейчас программа выполняется один раз и завершает работу. Представьте, что мы хотим, чтобы программа выполнялась больше одного раза, скажем, пять раз. В частности, давайте выполним «код проверки минуты» и инструкцию `if/else` пять раз, делая паузу на случайное количество секунд перед каждой следующей попыткой (просто, чтобы поддержать интерес). Когда программа завершится, на экране должны отображаться пять сообщений, а не одно.

Вот этот код, который нам нужно запустить множество раз в цикле:

Давайте заставим программу выполнять этот код несколько раз.

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Что нам надо сделать:

1. **Заключить код в цикл.**
   Цикл позволяет снова и снова выполнять любой блок кода, и Python предоставляет для этого несколько способов. В данном случае (не вдаваясь в почему) мы будем использовать цикл `for`.

2. **Приостановить выполнение на время.**
   В стандартном модуле `time` имеется функция с именем `sleep`, которая приостанавливает выполнение программы на заданное количество секунд.

3. **Сгенерировать случайное число.**
   К счастью, в другом модуле, с именем `random`, имеется функция `randint`, которую можно использовать для генерации случайного числа. Воспользуемся функцией `randint`, чтобы сгенерировать число в диапазоне от 1 до 60, и используем его для приостановки программы в каждой итерации цикла.

Теперь мы знаем, что мы хотим сделать. Но есть ли предпочтительный способ внесения этих изменений?

Как лучше подойти к решению этой проблемы?

В Python работают оба подхода

В Python можно следовать *обоим* этим подходам, но большинство программистов на Python предпочитают **экспериментировать**, пытаясь понять, какой код лучше подойдет для каждой конкретной ситуации.

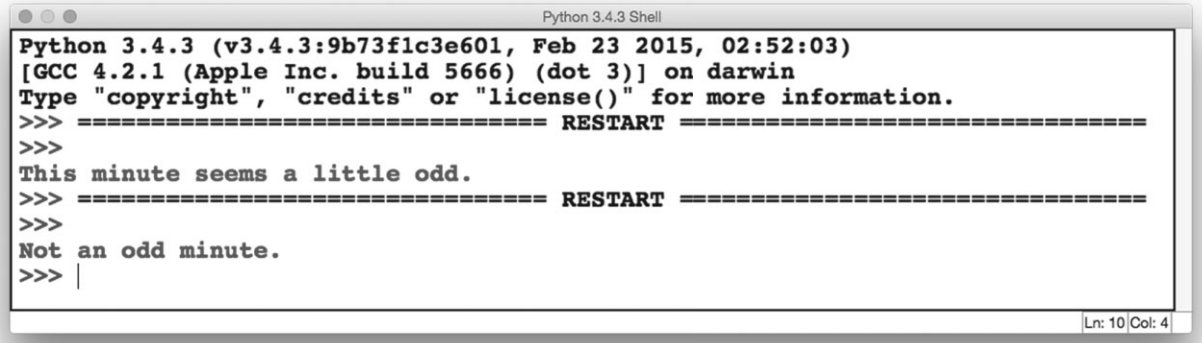
Не поймите нас неправильно: мы не утверждаем, что подход Макса является неверным, а подход Анны правильный. Просто в Python программистам доступны оба варианта, и оболочка Python (с которой мы по-быстрому ознакомились в начале главы) делает эксперименты естественным выбором.

Давайте определим, какой код нам нужен для расширения нашей программы, экспериментируя в командной строке >>>.

Эксперименты в командной строке >>> помогут вам разработать требуемый код.

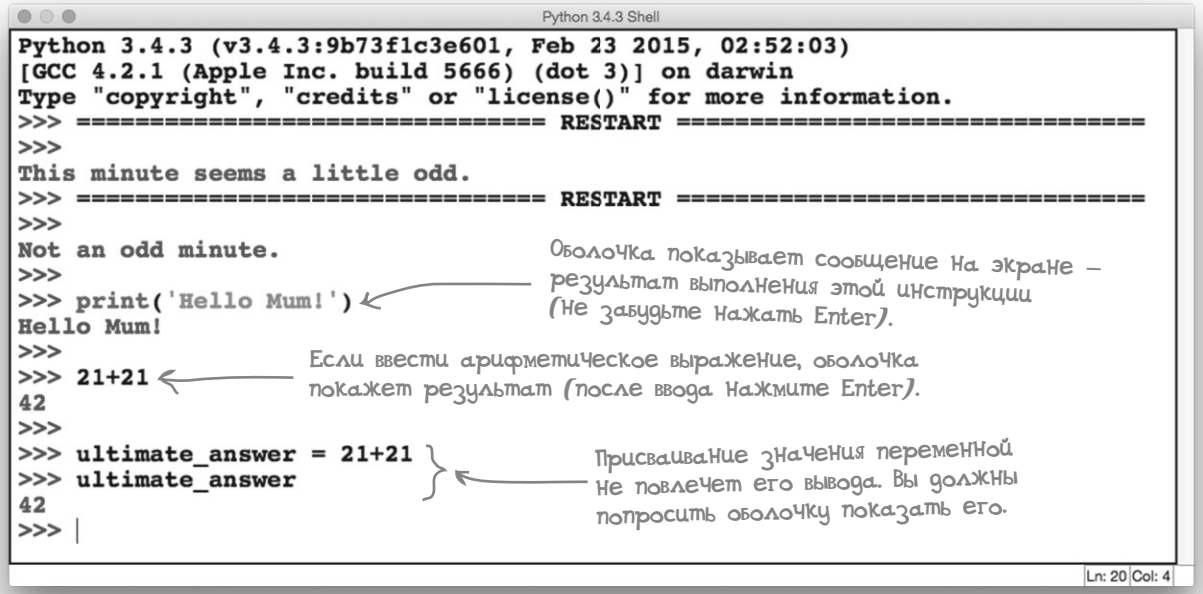
Возвращение в командную оболочку Python

Вот как командная оболочка Python выглядела в последний раз, когда мы взаимодействовали с ней (у вас она может выглядеть немного по-другому, так как сообщения могли появиться в другом порядке).



Командная оболочка Python (для краткости — просто «оболочка») показывает сообщения, выводимые нашей программой, но она может намного больше. Командная строка >>> позволяет вводить любые инструкции на языке Python и *немедленно* выполнять их. Если инструкция порождает вывод, оболочка покажет его. Если результатом инструкции является значение, оболочка покажет вычисленное значение. Однако если вы создаете новую переменную и присваиваете ей значение, вам придется ввести имя переменной в командную строку >>>, чтобы увидеть, какое значение в ней хранится.

Посмотрите пример взаимодействия с оболочкой, который показан ниже. Лучше, если вы будете следовать вместе с нами и попробуете эти примеры в *своей* оболочке. Ввод каждой инструкции обязательно завершайте нажатием клавиши *Enter*, чтобы сообщить оболочке, что она должна выполнить ее *сейчас же*.



Экспериментируем в оболочке

Теперь, когда вы знаете, как ввести одну инструкцию в командную строку >>> и сразу выполнить ее, вы можете начать создавать код, необходимый для расширения вашей программы.

Ниже перечислено, что должен делать новый код.

- ☐ **Повторять** выполнение заданное число раз. Мы уже решили использовать здесь встроенный цикл `for`.
- ☐ **Приостанавливать** выполнение программы на заданное количество секунд. Это может сделать функция `sleep` из модуля `time` стандартной библиотеки.
- ☐ **Генерировать** случайное число из заданного диапазона. Функция `randint` из модуля `random` позволяет проделать этот трюк.

Мы больше не будем показывать вам полные скриншоты IDLE и ограничимся только командной строкой >>> и выводом. В частности, с этого момента вместо скриншотов вы будете видеть что-то вроде этого:

На следующих нескольких страницах мы займемся экспериментами, чтобы выяснить, как добавить три возможности, перечисленные выше. Мы будем *играть* с кодом в командной строке >>>, пока не определим точные инструкции для добавления в нашу программу. Отложим пока код `odd.py` и активизируем окно оболочки, выбрав его. Курсор должен мигать справа от >>>, ожидая ввода.

Переверните страницу, когда будете готовы. Давайте же начнем эксперименты.

Перебор последовательности объектов

Мы ранее говорили, что будем использовать цикл `for`. Цикл `for` *отлично* подходит для управления итерациями, когда их количество известно заранее. (Если количество итераций неизвестно, мы рекомендуем использовать цикл `while`, но обсудим детали этого альтернативного способа организации цикла потом, когда он нам действительно понадобится). На этом этапе нам нужен `for`, поэтому посмотрим его действие в командной строке >>>.

Используйте «for», когда количество повторений известно заранее.

Существует три типичных сценария использования `for`. Давайте определим, какой из них подходит нам больше.

Пример использования 1. Цикл `for`, представленный ниже, принимает список чисел и перебирает его по одному элементу, выводя текущее значение на экран. Вот как это происходит: цикл `for` в каждой итерации присваивает число из списка *переменной цикла*, которой в этом коде дано имя `i`.

Поскольку код занимает больше одной строки, оболочка автоматически добавляет отступ, когда вы нажимаете клавишу Enter после двоеточия. Сигналом конца ввода для оболочки является *двойное нажатие* клавиши Enter в конце блока кода:

```
>>> for i in [1, 2, 3]:
        print(i)

1
2
3
```

Мы использовали «i» как переменную цикла, но вы могли бы назвать ее иначе. Кстати, «i», «j», и «k» невероятно популярны у программистов в такой ситуации.

Поскольку это блок кода, вам надо нажать клавишу Enter после его ввода ДВАЖДЫ, чтобы завершить инструкцию и выполнить ее.

Обратите внимание на *отступы* и *двоеточие*. Как и в случае с выражением `if`, код, ассоциированный с инструкцией `for`, должен быть **выделен отступами**.

Пример использования 2. Цикл `for`, представленный ниже, проходит по строке, каждый символ которой обрабатывается в отдельной итерации. Это возможно потому, что строки в Python являются **последовательностями**. Последовательность — это упорядоченная коллекция объектов (в этой книге мы увидим множество примеров последовательностей), и интерпретатор Python способен выполнять итерации через любые последовательности.

Последовательность — это упорядоченная коллекция объектов.

```
>>> for ch in "Hi!":
        print(ch)

H
i
!
```

Python понимает, что обработка этой строки должна происходить по одному символу за раз. Именно поэтому мы дали переменной цикла имя «ch» (от англ. «char» — «символ»).

Вы нигде не должны говорить циклу `for`, *насколько велика эта строка*. Python сам может понять, где *заканчивается* строка, и завершает цикл `for`, когда закончатся все объекты в последовательности.

Повторяем определенное количество раз

В цикл `for` можно передать не только последовательность для перебора, но также указать точное число итераций с помощью встроенной функции `range`.

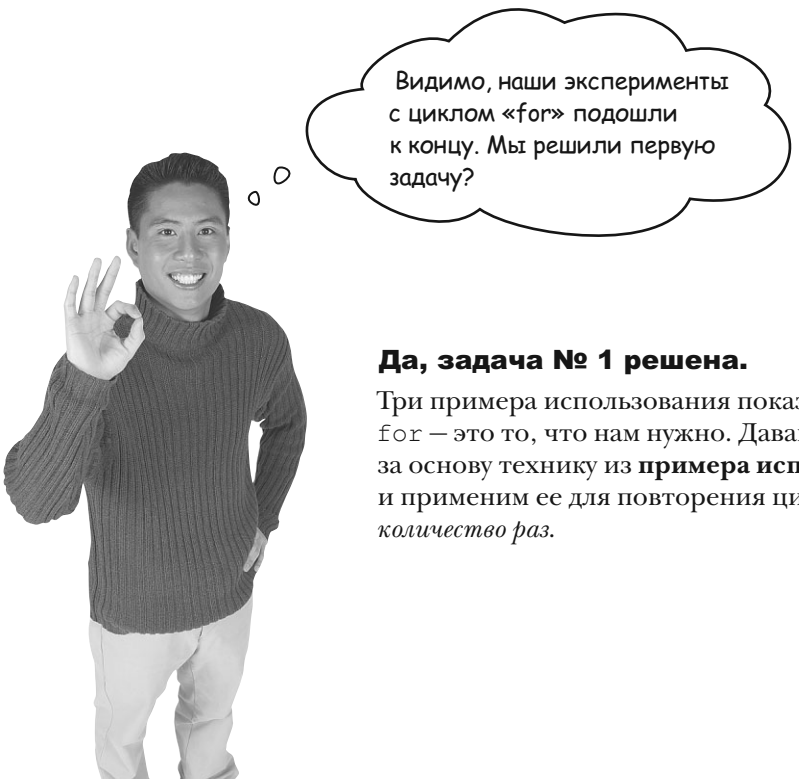
Давайте рассмотрим другой пример, демонстрирующий применение `range`.

Пример использования 3. В наиболее общем случае функции `range` передается один целочисленный аргумент, определяющий количество итераций (далее в книге мы увидим другие области применения `range`). В этом цикле мы используем `range`, чтобы сгенерировать список чисел, которые поочередно присваиваются переменной `num`.

```
>>> for num in range(5):
        print('Head First Rocks!')

Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
```

Мы попросили у range вернуть пять последовательных чисел, поэтому цикл выполнится пять раз, в результате чего появятся пять сообщений. Запомните: Нажимайте Enter дважды для запуска кода, в котором есть блок.

Переменная цикла `num` *нигде не используется* в теле цикла `for`. Но это не считается ошибкой, потому что только вы (как программист) можете решать, нужно ли использовать `num` в теле цикла. В данном случае с `num` ничего делать не надо.

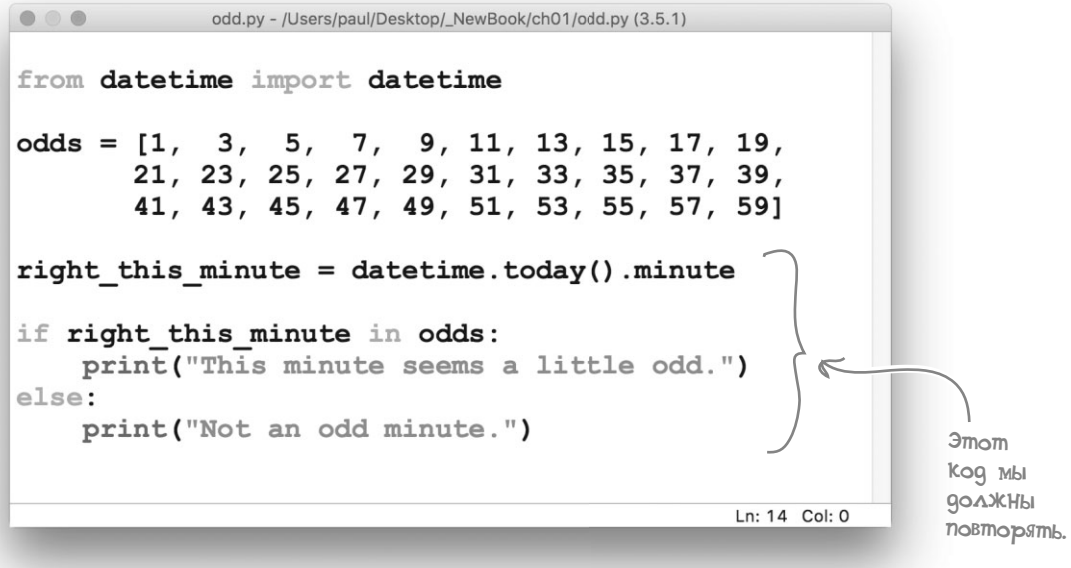


Да, задача № 1 решена.

Три примера использования показали, что цикл `for` — это то, что нам нужно. Давайте возьмем за основу технику из **примера использования 3** и применим ее для повторения цикла *заданное количество раз*.

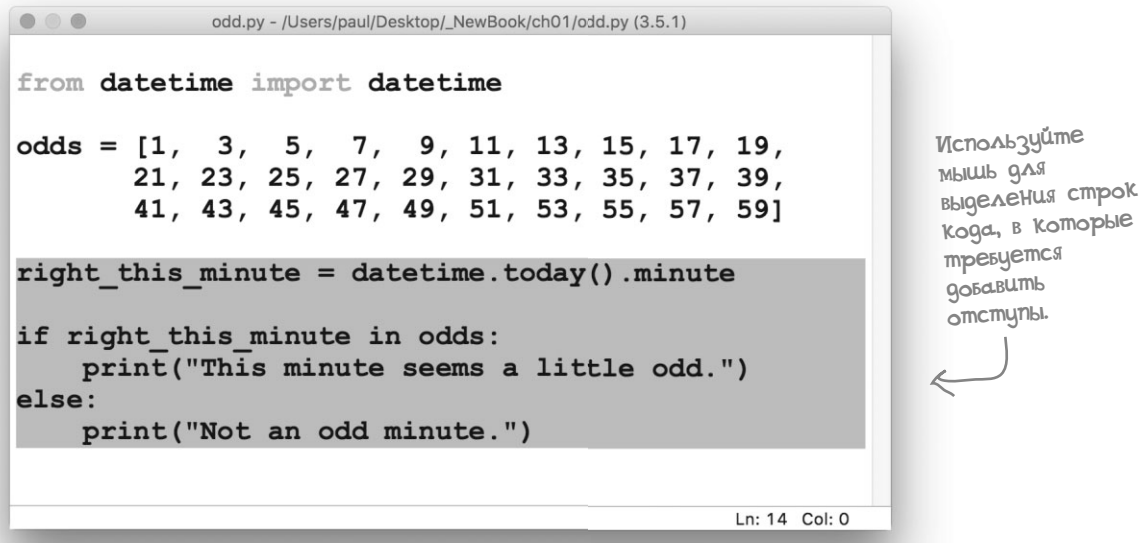
Применим решение задачи № 1 к нашему коду

Вот как наш код выглядел в окне редактирования IDLE *до того*, как мы решили задачу № 1.



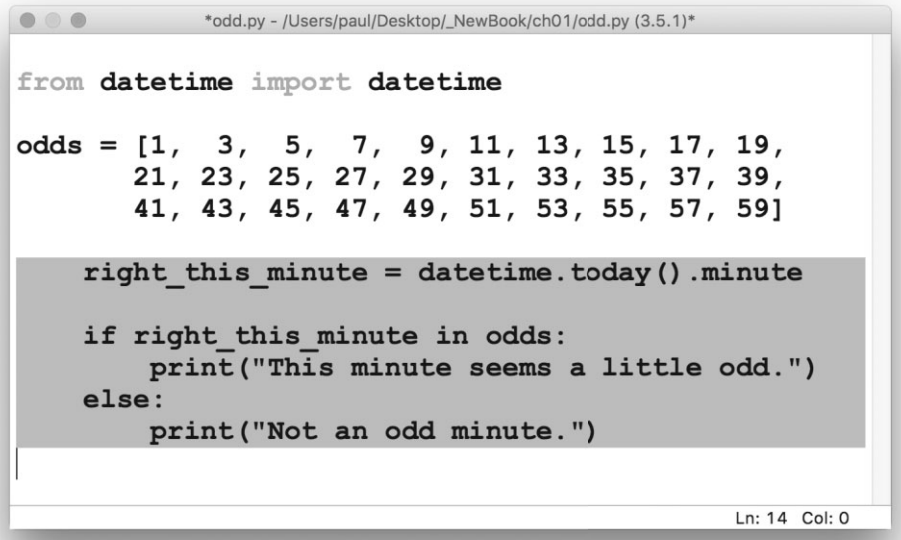
Теперь вы знаете, что последние пять строк в программе можно повторить пять раз с помощью цикла `for`. В эти пять строк следует **добавить отступы**, чтобы превратить их в блок кода цикла `for`. А именно: каждую строку кода нужно сместить вправо на *один* отступ. Но не тратьте силы на выполнение этого действия с каждой строкой. Позвольте IDLE добавить отступы сразу во все строки за *один раз*.

Выделите мышью строки кода, в которые требуется добавить отступы.



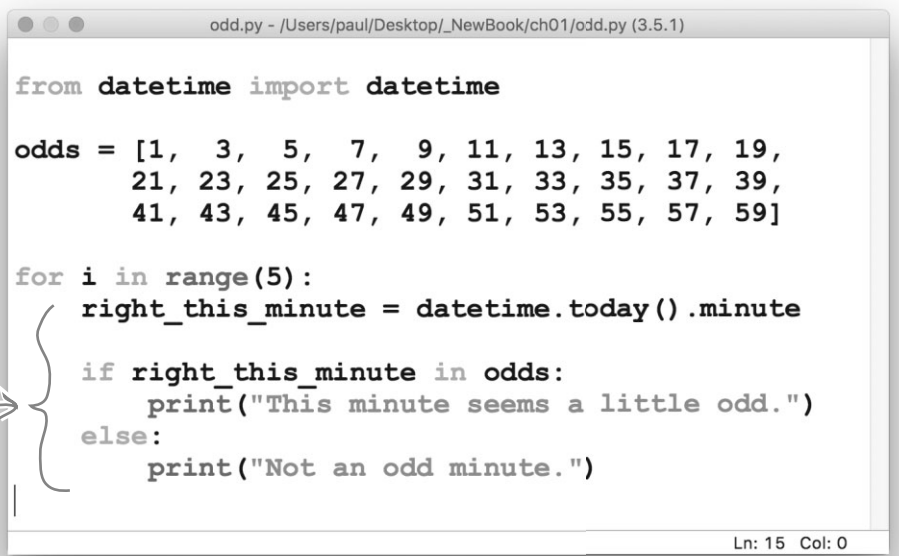
Оформление отступов с Format...Indent Region

Выделив пять строк кода, выберите пункт *Indent Region* в меню *Format* в окне редактирования IDLE. Выделенный блок целиком сместится вправо на один отступ:



Пункт Indent Region в меню Format добавит отступ во все выделенные строки кода за раз.

Заметьте также, что в IDLE имеется пункт меню *Dedent Region*, убирающий отступ из выделенного блока, и обе команды — *Indent* и *Dedent* — имеют горячие клавиши, отличающиеся в разных операционных системах. Найдите время на изучение горячих клавиш, которые используются в вашей системе, *сейчас* (поскольку вы будете использовать их постоянно). После оформления отступа в блоке кода можно добавить цикл `for`:



Добавляем строку с заголовком цикла «for».

Блок кода цикла «for» правильно оформлен отступами.

Устраиваем паузу выполнения

Давайте вспомним, что нам нужно сделать.

- [x] **Повторить** заданное количество раз.
- [ ] **Приостановить** программу на заданное количество секунд.
- [ ] **Сгенерировать** случайное число между двумя заданными значениями.

Теперь вернемся в оболочку и опробуем некоторый код, который поможет нам со второй задачей: *приостановить программу на заданное количество секунд.*

Однако прежде вспомним первую строку в нашей программе, которая импортирует заданную функцию из заданного модуля.

```
from datetime import datetime
```

При таком использовании «import» добавляет указанную функцию (или субмодуль, как в данном примере) в программу. После этого к функции (или к субмодулю) можно обращаться без применения синтаксиса точечной нотации.

Это один способ импортировать функцию в программу. Другой, столь же распространенный прием, — импортировать модуль *без* определения функции, которую вы хотите использовать. Давайте применим второй способ, так как он будет появляться во многих программах на Python, которые вам встретятся.

Как упоминалось ранее в главе, приостановить выполнение программы на заданное количество секунд можно вызовом функции `sleep`, которая находится в модуле `time` из стандартной библиотеки. Давайте *сначала* **импортируем** модуль, пока без упоминания `sleep`.

```
>>> import time
>>>
```

Эта инструкция требует от оболочки импортировать модуль «time».

Когда инструкция `import` используется, как в примере с модулем `time`, показанном выше, вы получаете доступ к возможностям модуля только по его *имени*, импортированному в код вашей программы. Чтобы получить доступ к функции в модуле, импортированном таким способом, используйте синтаксис точечной нотации, как показано здесь.

Первым следует имя модуля (перед точкой).

Это продолжительность паузы в секундах.

```
>>> time.sleep(5)
>>>
```

Имя вызываемой функции (после точки).

Заметьте, что такой вызов `sleep` приостановит оболочку на пять секунд, и только потом вновь появится командная строка >>>. Вперед, и *попробуйте это сейчас.*

Важная путаница

Подождите секундочку... Python поддерживает два механизма импортирования? Разве это не приводит к путанице?

Отличный вопрос.

Для ясности: в Python не *два* механизма импортирования, так как есть только *одна* инструкция `import`. Однако инструкцию `import` можно использовать *двумя способами*.

Первый, который мы изначально видели в примере программы, импортирует именованную функцию в **пространство имен** нашей программы, что позволило нам вызывать функцию по мере необходимости без *связывания* функции с импортированным модулем. (Понятие пространства имен очень важно в Python, поскольку оно определяет контекст выполнения кода. И тем не менее мы подождем до следующей главы, где детально исследуем пространства имен.)

В нашем примере программы мы использовали первый способ импортирования, а затем вызывали функцию `today()` как `datetime.today()`, а *не* как `datetime.datetime.today()`.

Второй способ использования `import` — это просто импортирование модуля, как мы делали это, когда экспериментировали с модулем `time`. Импортировав модуль этим способом, мы должны использовать синтаксис точечной нотации для доступа к функциям этого модуля, как мы это сделали с `time.sleep()`.

Это не Глупые вопросы

В: Есть правильный способ использования `import`?

О: Это часто определяется личными предпочтениями, так как некоторые программисты любят быть очень конкретными, в то время как другие — нет. Однако случаются ситуации, когда два модуля (будем называть их `A` и `B`) содержат функции с одинаковыми именами, назовем их `F`. Если в своей программе вы напишете две инструкции импортирования: «`from A import F`» и «`from B import F`», то как Python узнает, какую `F` надо выполнить, когда вы вызовете `F()`? Единственный способ избежать неоднозначности — использовать неспецифицированную инструкцию `import` (то есть поместить в программу инструкции «`import A`» и «`import B`»), а затем вызывать нужную функцию F с использованием полного квалифицированного имени: `A.F()` или `B.F()`. Это сведет путаницу на нет.

Генерация случайных чисел на Python

Хотя и заманчиво добавить `import time` в начало нашей программы, а затем вызывать `time.sleep(5)` в теле цикла `for`, мы не будем делать этого сейчас. Мы не закончили с экспериментами. Приостановки на пять секунд недостаточно; нам надо иметь возможность приостанавливать на *случайный временной интервал*. С мыслями об этом давайте напомним себе, что мы уже сделали и что нам осталось.

- ☑ **Повторить** заданное количество раз.
- ☑ **Приостановить** программу на заданное количество секунд.
- ☐ **Сгенерировать** случайное число между двумя заданными значениями.

Решив эту последнюю задачу, мы сможем вернуться и уверенно изменить нашу программу, чтобы включить в нее все, что мы изучили в ходе экспериментов. Но мы пока не готовы, поэтому давайте рассмотрим последнюю задачу: получение случайного числа.

Как в случае с приостановкой, решить эту задачу нам поможет *стандартная библиотека*, а точнее имеющийся в ней модуль `random`. Основываясь всего лишь на этой скудной информации, перейдем к экспериментам в оболочке.

```
>>> import random
>>>
```

Используйте «dir», чтобы получить информацию об объекте.

Что теперь? Мы могли бы заглянуть в документацию или обратиться к справочнику... но для этого нам придется отвлечься от оболочки, даже если это займет всего несколько минут. Как оказывается, оболочка предоставляет некоторые дополнительные функции, которые могут помочь нам. Эти функции не предназначены для использования в программах; они создавались для командной строки >>>. Первая называется `dir`, и она показывает **все атрибуты**, связанные с чем-нибудь, включая модули:

Имя нужной нам функции спрятано в середине этого длинного списка.

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF',
'Random',    ...    'randint', 'random', 'randrange',
'sample', 'seed', 'setstate', 'shuffle', 'triangular',
'uniform', 'vonmisesvariate', 'weibullvariate']
```

Это сокращенный список. Тот, что вы увидите на своем экране, будет намного длиннее.

Этот список содержит много чего. Но нас интересует функция `randint()`. Чтобы узнать больше о `randint`, давайте попросим **помощи** у оболочки.

Просим у интерпретатора помощи

Узнав имя чего-либо, можно попросить **помощи** у оболочки. Когда вы это делаете, оболочка показывает раздел из документации по Python, относящийся к имени, которое вас интересует (здесь и далее текст справки дан в переводе. — *Прим. науч. ред.*).

Используйте «help», чтобы получить описание.

Давайте посмотрим, как действует этот механизм в командной строке >>>, и попробуем получить **помощь** по функции `randint` из модуля `random`:

Просим помощи в командной строке >>>...

```
>>> help(random.randint)
Справка по методу randint из модуля random:

randint(a, b) метод экземпляра.Random instance
    Возвращает случайное целое число в диапазоне [a, b],
    включая оба граничных значения.
```

...и видим связанную документацию прямо в оболочке.

Для умников

Вы можете повторно вызвать последнюю команду(ы), нажав в командной строке IDLE >>> комбинацию Alt-P, если вы работаете в Linux или Windows. В Mac OS X используйте комбинацию Ctrl-P. Думайте, что «P» означает «previous» («предыдущие»).

Быстрое чтение описания функции `randint` подтверждает, что это именно то, что нам нужно: если мы передадим ей два определенных значения, обратно получим случайное целое число из заданного диапазона.

Несколько заключительных экспериментов в командной строке >>> демонстрируют функцию `randint` в действии.

```
>>> random.randint(1,60)
27
>>> random.randint(1,60)
34
>>> random.randint(1,60)
46
```

Если вы следовали за нами, вы увидите другие результаты, так как целые числа, возвращаемые «randint», генерируются случайно.

Поскольку вы импортировали модуль «random» инструкцией «import random», не забывайте предварять вызов «randint» именем модуля с точкой. Вот так: «random.randint()», а не «randint()».

После этого вы можете поставить галочку напротив последней задачи, поскольку теперь вы знаете достаточно, чтобы получить случайное число в заданном диапазоне:

- [x] **Сгенерировать** случайное число между двумя заданными значениями.

Пора вернуться к нашей программе и внести изменения.

Обзор наших экспериментов

Перед тем как идти вперед и менять программу, сделаем краткий обзор результатов наших экспериментов в оболочке.

Сначала мы написали цикл `for`, который повторяется пять раз.

```
>>> for num in range(5):
        print('Head First Rocks!')

Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
Head First Rocks!
```

Мы запросили последовательность из пяти чисел, поэтому было произведено пять итераций, результатом которых стали пять сообщений.

Затем мы использовали функцию `sleep` из модуля `time`, чтобы приостановить выполнение программы на заданное количество секунд.

```
>>> import time
>>> time.sleep(5)
```

Импортировав модуль «time», мы получили возможность вызывать функцию «sleep».

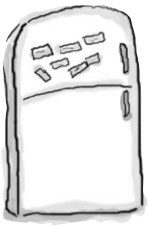
Затем мы поэкспериментировали с функцией `randint` (из модуля `random`) и сгенерировали несколько случайных чисел из заданного диапазона.

```
>>> import random
>>> random.randint(1,60)
12
>>> random.randint(1,60)
42
>>> random.randint(1,60)
17
```

Заметьте: каждый раз генерируются разные числа, потому что каждый вызов «randint» возвращает разные случайные числа.

Теперь мы можем собрать все вместе и изменить нашу программу.

Давайте напомним себе, что мы решили ранее в этой главе: организовать выполнения нашей программы в цикле, выполнять «код проверки минуты» и инструкцию `if/else` пять раз и приостанавливать выполнение на случайное количество секунд между итерациями. В результате на экране должно появиться пять сообщений.



Магнитики с экспериментальным кодом

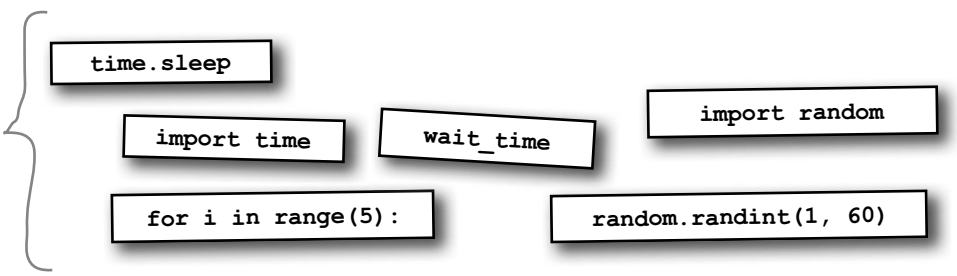
Основываясь на описании в нижней части предыдущей страницы, а также на результатах наших экспериментов, мы забежали вперед и сделали всю необходимую работу за вас. Но после того как мы разместили наши магнитики с кодом на холодильнике, кто-то (не спрашивайте кто) хлопнул дверью, и сейчас часть нашего кода оказалась на полу.

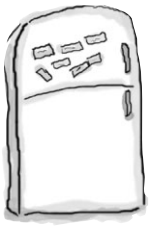
Ваша задача заключается в том, чтобы вернуть их назад так, чтобы мы могли запустить новую версию нашей программы и подтвердить, что она работает, как требуется.

Решите, какие магнитики кода должны попасть на место каждой из пунктирных линий.

```
from datetime import datetime

.....
.....

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

.....
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = .....
    ..... ( ..... )
```

Куда все они подойдут?





Решение для магнитиков с кодом

Основываясь на описании выше, а также на результатах наших экспериментов, мы забежали вперед и сделали всю необходимую работу за вас. Но после того как мы разместили наши магнитики с кодом на холодильнике, кто-то (не спрашивайте кто) хлопнул дверью, и сейчас часть нашего кода оказалась на полу.

Ваша задача заключалась в том, чтобы вернуть их назад так, чтобы мы могли запустить новую версию нашей программы и подтвердить, что она работает, как требуется.

Импортирование необязательно выполнять в начале программы, но это устоявшееся соглашение в среде программистов на Python.

```
from datetime import datetime
import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]
```

Цикл «for» повторяется ТОЧНО пять раз.

```
for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

Весь этот код под инструкцией «for» оформлен отступами, потому что является частью блока кода «for». Запомните: Python не использует фигурные скобки для выделения блоков кода; вместо этого используются отступы.

Функция «randint» возвращает случайное целое число, которое присваивается новой переменной с именем «wait_time», которая...

...затем используется в вызове «sleep» для приостановки программы на случайное количество секунд.

Пробная поездка

Давайте попробуем запустить нашу обновленную программу в IDLE и посмотрим, что случится. Измените вашу версию `odd.py`, как требуется, сохраните копию вашей новой программы как `odd2.py`. Когда будете готовы, нажмите клавишу F5, чтобы выполнить код.

Когда вы нажмете F5, чтобы запустить код...

odd2.py - /Users/Paul/Desktop/_NewBook/ch01/odd2.py (3.4.3)

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

Ln: 19 Col: 0

...вы должны увидеть вывод, похожий на этот. Только помните, что ваш вывод будет отличаться, так как случайные числа, которые сгенерирует ваша программа, почти наверняка не совпадут с нашими.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
This minute seems a little odd.
This minute seems a little odd.
Not an odd minute.
Not an odd minute.
Not an odd minute.
>>>
```

Ln: 25 Col: 4

Не беспокойтесь, если вы увидите список сообщений, отличающийся от показанного здесь. Главное, чтобы сообщений было ровно пять, в соответствии с количеством итераций цикла.

Обновим то, что мы уже знаем

Имея работающую программу `odd2.py`, остановимся еще раз, чтобы повторить все новое, что мы узнали о Python на последних 15 страницах.

КОНТРОЛЬНЫЙ СПИСОК

- При попытке подобрать код для решения той или иной проблемы программисты на Python часто предпочитают экспериментировать с фрагментами кода в оболочке.
- Если вы видите командную строку `>>>`, вы в оболочке Python. Продолжайте: введите инструкцию на языке Python и посмотрите, что случится, когда она выполнится.
- Оболочка принимает ваши строки кода и посылает их интерпретатору, который затем выполняет их. Все результаты возвращаются в оболочку и выводятся на экран.
- Цикл `for` можно использовать для выполнения фиксированного количества итераций. Если число повторений известно заранее, используйте `for`.
- Когда число повторений заранее неизвестно, используйте цикл `while` (с которым нам еще предстоит познакомиться, но не беспокойтесь: мы еще увидим, как он действует).
- Цикл `for` может перебирать элементы последовательностей (таких как списки или строки), а также выполняться фиксированное количество раз (благодаря функции `range`).
- Если потребуется приостановить выполнение программы на заданное количество секунд, используйте функцию `sleep` из модуля `time` стандартной библиотеки.
- Вы можете импортировать отдельную функцию из модуля. Например, `from time import sleep` импортирует функцию `sleep`, после чего ее можно будет вызывать без квалификации именем модуля.
- Если вы просто импортируете модуль, например `import time`, тогда при вызове любых функций из модуля их нужно квалифицировать именем модуля, например: `time.sleep()`.
- В модуле `random` есть очень полезная функция `randint`, которая генерирует случайное целое число в заданном диапазоне.
- Оболочка предоставляет две интерактивные функции, которые работают в командной строке `>>>`. Функция `dir` перечисляет все атрибуты объекта, а `help` предоставляет доступ к документации по Python.

Это не глупые вопросы

В: Должен ли я запомнить все это?

О: Нет, и не заставляйте себя, если ваш мозг сопротивляется запоминанию всего увиденного. Это только первая глава, и мы создавали ее как краткое введение в мир программирования на Python. Если вы понимаете суть того, что происходит с кодом, это уже хорошо.

Несколько строк кода делают многое

Так и есть, но мы отлично справляемся.

В действительности мы затронули лишь малую часть языка Python. Но то, что мы увидели, было очень полезным.

То, что мы видели до сих пор, помогает показать одну из сильных сторон Python: *несколькими строками кода можно сделать многое*. Другая сторона языка, прославившая его: *код на Python легко читать*.

В попытке доказать, насколько это легко, мы представляем на следующей странице совершенно другую программу, для понимания которой вы уже достаточно знаете о Python.

Не желаете бутылочку хорошего, холодного пива?

Создание серьезного бизнес-приложения

99 бутылок пива на стене
99 бутылок пива!
Возьми одну, пусти по кругу
98 бутылок пива на стене!

Отдавая дань уважения книге «Изучаем Java», рассмотрим версию классического первого серьезного приложения на Python — песню про пиво.

Ниже на скриншоте показан код Python-версии песни про пиво (https://ru.wikipedia.org/wiki/99_бутылок_пива). Кроме отличия в использовании функции `range` (которое мы обсудим в двух словах), большая часть кода должна быть понятной. Окно редактирования IDLE содержит код, а в окне оболочки видно окончание вывода программы:

beersong.py - /Users/Paul/Desktop/_NewBook/ch01/beersong.py (3.4.3)

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on the wall.")
    print(beer_num, word, "of beer.")
    print("Take one down.")
    print("Pass it around.")
    if beer_num == 1:
        print("No more bottles of beer on the wall.")
    else:
        new_num = beer_num - 1
        if new_num == 1:
            word = "bottle"
        print(new_num, word, "of beer on the wall.")
    print()
```

Выполнение этого кода дает такой результат в оболочке.

Python 3.4.3 Shell

```
3 bottles of beer on the wall.
3 bottles of beer.
Take one down.
Pass it around.
2 bottles of beer on the wall.

2 bottles of beer on the wall.
2 bottles of beer.
Take one down.
Pass it around.
1 bottle of beer on the wall.

1 bottle of beer on the wall.
1 bottle of beer.
Take one down.
Pass it around.
No more bottles of beer on the wall.

>>>
```

Ln: 660 Col: 12

Подсчет бутылок пива...

Код выше, введенный в окно редактирования IDLE и сохраненный, после нажатия F5 производит много вывода в оболочку. Мы показали только небольшой фрагмент полученного вывода в окне справа, так как песня начиналась с 99 бутылок пива, количество которых уменьшалось, пока пива больше не осталось. Фактически в этом коде только один по-настоящему интересный момент — это обработка «обратного отсчета», так что давайте посмотрим, как это работает, прежде чем переходить к подробному обсуждению кода программы.

Код Python легко читается

1 бутылка пива На стене
1 бутылка пива!
Возьми одну, пусти по кругу
Нет бутылок пива На стене!

Нет бутылок пива На стене!
Нет бутылок пива!
Пойди в магазин и купи ещё.
99 бутылок пива На стене!

А ни в чем!

Большинство программистов при первом столкновении с кодом на Python, таким как в песне про пиво, предполагают, что должно что-то случиться где-то еще.

Там должен быть подвох, но его нет?

Нет, его нет. Это не случайно, что код на Python легко читается: язык разработан с учетом этой конкретной цели. Гвидо ван Россум (Guido van Rossum), создатель языка, хотел создать мощный инструмент программирования, производящий код, который будет легко поддерживать, а это подразумевает, что код на языке Python должен также легко читаться.

Для умников

Чтобы использовать кириллицу, недостаточно заменить строку "bottles" строкой "бутылок". Для правильного склонения существительного 'бутылка' по числам придется две строки:

```
if new_num == 1:
     word = "bottle"
```

заменить строками:

```
if new_num >= 11 and new_num <= 19:
     word = "бутылок"
else:
     if new_num % 10 == 1:
          word = "бутылка"
     elif new_num % 10 in (2, 3, 4):
          word = "бутылки"
     else:
          word = "бутылок"
```

При использовании текстового редактора, сохраняющего файлы в кодировке, отличной от UTF-8, в начало программы следует вставить директиву:

```
# -*- coding: cp866 -*-
```

или

```
# -*- coding: cp1251 -*-
```

в зависимости от того, в какой кодировке сохраняется текст программы (прим. научного редактора).

Отступы вас бесят?

Подождите секундочку.
Все эти отступы меня бесят.
В этом подвох?

Чтобы привыкнуть к отступам, нужно время.

Не беспокойтесь. Все приходящие в Python от «фигурно-скобочных языков» *поначалу* борются с отступами. Но потом справляются с этим. Через день или два работы с Python вы вряд ли заметите, как оформляете свои блоки отступами.

Одна из проблем, возникающая у некоторых программистов с отступами, вызвана тем, что они смешивают *табуляции* с *пробелами*. Так как интерпретатор не смотрит на внешний вид кода, а подсчитывает **пробельные символы в отступах**, это может привести к таким проблемам, когда код «выглядит хорошо», но отказывается работать. Это неприятно, когда вы начинаете программировать на Python.

Наш совет: *не смешивайте табуляции и пробелы в коде на Python*.

Более того, мы советуем вам настроить свой редактор, чтобы нажатие клавиши *Tab* он заменял *четырьмя пробелами* (а заодно и автоматически удалял пробельные символы в конце строки). Это также общепринятая практика среди множества программистов на Python, и вы тоже должны следовать ей. Мы расскажем больше о работе с отступами в конце этой главы.

Возвращаемся к коду песни про пиво

Если вы посмотрите на вызов `range` в песне про пиво, то заметите, что он принимает *три* аргумента, а не один (как в нашем первом примере программы).

Присмотритесь к коду, но пока не заглядывайте в объяснение на следующей странице. Сможете ли вы сами разобраться, что возвращает такой вызов `range`:

beersong.py - /Users/Paul/Desktop/_NewBook/ch0...

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on
    print(beer_num, word, "of beer.")
    print("Take one down.")
```

Это что-то новое: вызов «range» принимает три аргумента, а не один.

Попросим интерпретатор помочь с функцией

Напомним, что вы можете обратиться к оболочке за **справкой** о чем угодно, что касается языка Python, поэтому давайте попросим справку о функции `range`.

Если сделать это в IDLE, описание не уместится на одном экране и быстро прокрутится вверх. Поэтому вам надо прокрутить текст в окне обратно, до строки, где вы попросили у оболочки помощи (и где находится самое интересное место в описании `range`).

```
>>> help(range)
Справка о классе range в модуле builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Возвращает последовательность чисел от start до stop
с шагом step.
           ...
```

Функцию «range» можно вызвать двумя способами.

Похоже, это то, что нам нужно.

Начальное, конечное и шаг

Поскольку `range` — не единственное место, где вам встретятся **начальное значение**, **конечное значение** и **шаг**, воспользуемся моментом, чтобы описать, что означает каждое из них, а затем рассмотрим некоторые типичные примеры (на следующей странице).

1. **НАЧАЛЬНОЕ ЗНАЧЕНИЕ определяет, ОТКУДА начинается диапазон.**
   До сих пор мы использовали версию `range` с одним аргументом, который — как следует из документации — должен быть **конечным значением**. Когда другие значения опущены, `range` по умолчанию использует `0` в качестве **начального значения**, но вы можете передать другое значение по своему выбору. Поступая так, вы *должны* указать также **конечное значение**. В этом случае `range` получает несколько аргументов при вызове.

2. **КОНЕЧНОЕ ЗНАЧЕНИЕ определяет, ГДЕ закончится диапазон.**
   Мы уже использовали это значение, когда вызывали `range(5)` в нашем коде. Заметьте, что генерируемая последовательность *никогда* не включает **конечного значения**, так что это до **конечного значения, но не включая его**.

3. **ШАГ определяет, КАК генерируется диапазон.**
   Когда определены **начальное** и **конечное** значения, можно также (опционально) определить значение **шага**. По умолчанию **шаг** равен 1 и требует от `range` генерировать диапазон с *шагом* `1`; то есть `0, 1, 2, 3, 4` и так далее. Вы можете установить любое значение **шага**. Можно даже установить отрицательное значение **шага**, чтобы изменить *направление* генерируемого диапазона.

Эксперименты с диапазонами

Теперь, когда вы узнали чуть больше о **начальном**, **конечном значении** и **шаге**, давайте поэкспериментируем в оболочке и посмотрим, как с помощью функции `range` производить разные диапазоны целых чисел.

Чтобы видеть результаты, мы используем другую функцию, `list`, для преобразования результата вызова `range` в читаемый список, который мы можем увидеть на экране:

```
>>> range(5)
range(0, 5)
```

Так мы использовали «range» в нашей первой программе.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Передадим вывод «range» функции «list», чтобы получить список.

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

Мы можем задать НАЧАЛЬНОЕ и КОНЕЧНОЕ значения для «range».

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Мы также можем задать значение ШАГА.

```
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

Становится действительно интересно, когда вы меняете направление диапазона, устанавливая отрицательное значение ШАГА.

```
>>> list(range(10, 0, 2))
[]
```

Python не убережет вас от глупости. Если ваше НАЧАЛЬНОЕ ЗНАЧЕНИЕ больше, чем КОНЕЧНОЕ, а ШАГ положителен, вы не получите ничего (в этом случае – пустой список).

```
>>> list(range(99, 0, -1))
[99, 98, 97, 96, 95, 94, 93, 92,  ...  5, 4, 3, 2, 1]
```

После всех экспериментов мы добрались до вызова `range` (показан выше последним), который создает список значений от 99 до 1, точно такой, как цикл `for` из песни о пиве.

beersong.py - /Users/Paul/Desktop/_NewBook/ch0...

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on
    print(beer_num, word, "of beer.")
    print("Take one down.")
```

Вызов «range» принимает три аргумента: начальное значение, конечное значение и шаг.

Заточите карандаш

Здесь снова приводится код программы с песней про пиво, который размазан по всей странице, чтобы вы могли **сосредоточиться** на каждой строке кода, составляющей это «серьезное бизнес-приложение».

Возьмите карандаш и в специально отведенных для этого местах напишите, что, по вашему мнению, делает каждая строка кода. Обязательно попытайтесь сделать это самостоятельно, прежде чем смотреть на то, что мы придумали на следующей странице. Мы начали за вас, написав, что делает первая строка.

```
word = "bottles"                                          Присваивает значение «bottles» (строку) новой переменной с именем «word».

for beer_num in range(99, 0, -1):

    print(beer_num, word, "of beer on the wall.")

    print(beer_num, word, "of beer.")

    print("Take one down.")

    print("Pass it around.")

    if beer_num == 1:

        print("No more bottles of beer on the wall.")

    else:

        new_num = beer_num - 1

        if new_num == 1:

            word = "bottle"

        print(new_num, word, "of beer on the wall.")

    print()
```

Заточите карандаш Решение

И снова код программы с песней про пиво, который размазан по всей странице, чтобы вы могли **сосредоточиться** на каждой строке кода, составляющей это «серьезное бизнес-приложение».

Вы взяли карандаш и затем, в специально отведенных для этого местах, написали, что, по вашему мнению, делает каждая строка кода. Мы начали за вас, указав, что делает первая строка кода.

У вас получилось? Ваши объяснения похожи на наши?

```
word = "bottles"
```

Присваивает значение «bottles» (строку) новой переменной с именем «word».

```
for beer_num in range(99, 0, -1):
```

Начинает цикл, выполняющийся заданное количество раз, от 99 до нуля. «beer_num» используется как переменная цикла.

```
    print(beer_num, word, "of beer on the wall.")
    print(beer_num, word, "of beer.")
    print("Take one down.")
    print("Pass it around.")
```

Четыре вызова функции print выводят на экран текст песни для текущей итерации «99 bottles of beer on the wall 99 bottles of beer. Take one down. Pass it around» («99 бутылок пива на стене. 99 бутылок пива. Возьми одну. Пусти по кругу.») и так далее на каждой итерации.

```
    if beer_num == 1:
```

Проверка: по кругу пущена последняя бутылка?

```
        print("No more bottles of beer on the wall.")
```

Если да, закончить текст песни.

```
    else:
```

Иначе...

```
        new_num = beer_num - 1
```

Запомнить количество бутылок пива для следующей итерации в другой переменной с именем «new_num».

```
        if new_num == 1:
```

Если мы собираемся выпить последнюю бутылку пива...

```
            word = "bottle"
```

Меняем «bottles» («бутылки»), хранящиеся в переменной «word», на «bottle» («бутылка»).

```
        print(new_num, word, "of beer on the wall.")
```

Завершаем слова песни для этой итерации.

```
    print()
```

В конце итерации печатаем пустую строку. Если все итерации завершены, завершаем программу.

Не забудьте опробовать программу с песней про пиво

Если вы еще не сделали этого, введите код программы с песней про пиво в IDLE, сохраните ее как `beersong.py`, а затем нажмите клавишу F5, чтобы запустить.
Не переходите к следующей главе, пока не увидите песню про пиво.

это не Глупые вопросы

В: Я постоянно получаю ошибки, когда пытаюсь запустить свой код песни про пиво. Но мой код выглядит хорошо, поэтому я немного разочарован. Какие будут предложения?

О: В первую очередь проверьте правильность расстановки отступов. Если вы это сделали, проверьте, не смешали ли вы табуляции с пробелами в своем коде. Запомните: код будет выглядеть отлично (для вас), но интерпретатор откажется выполнять его. Эту проблему можно быстро исправить, перенеся код в окно редактирования IDLE, а затем выбрав пункт меню *Edit…→ Select All* и следом пункт меню *Format…→ Untabify Region*. Если вы смешали табуляции с пробелами, эти две команды преобразуют все табуляции в пробелы за один раз (и исправят любые проблемы с отступами).

Теперь вы можете сохранить свой код и нажать клавишу F5, чтобы попытаться запустить его снова. Если он по-прежнему отказывается работать, убедитесь, что ваш код точно такой же, как и наш в этой главе. Будьте очень внимательны к орфографическим ошибкам, которые могут закрасться в имена переменных.

В: Интерпретатор Python не предупредит меня, если я ошибусь, написав `new_num` как `nwe_num`?

О: Нет. Встретив инструкцию присваивания значения переменной, Python полагает, что вы знаете, что делаете, и продолжает выполнять ваш код. За именами переменных придется следить вам, так что будьте внимательны.

Повторим то, что вы уже знаете

Вот некоторые новые вещи, которые вы узнали в ходе работы (и запуска) кода песни о пиве.

КОНТРОЛЬНЫЙ СПИСОК

- Для привыкания к отступам не требуется много времени. Каждый программист, новичок в Python, в какой-то момент жалуется на отступы, но не волнуйтесь: скоро вы даже не будете замечать, как делаете их.
- Если есть что-то, чего вы не должны делать, то это смешивать табуляции и пробелы в отступах. Уберегите себя от будущих страданий и не делайте так.
- Функция `range` может принимать более одного аргумента. Эти аргументы позволяют контролировать начальное и конечное значения диапазона, а также шаг.
- Шаг в вызове функции `range` может быть отрицательным и тем самым менять направление генерируемого диапазона.

Пиво закончилось, что дальше?

Глава 1 подходит к концу. В следующей главе вы больше узнаете о том, как Python обрабатывает данные. В этой главе мы едва коснулись темы **списков**, и пришло время нырнуть немного глубже.

Код из главы 1

```
from datetime import datetime

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

right_this_minute = datetime.today().minute

if right_this_minute in odds:
    print("This minute seems a little odd.")
else:
    print("Not an odd minute.")
```

Мы начали с программы «odd.py», затем...

...расширили ее и создали программу «odd2.py», которая запускает «код проверки минуты» пять раз (благодаря встроенному в Python циклу «for»).

```
from datetime import datetime

import random
import time

odds = [ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19,
        21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
        41, 43, 45, 47, 49, 51, 53, 55, 57, 59 ]

for i in range(5):
    right_this_minute = datetime.today().minute
    if right_this_minute in odds:
        print("This minute seems a little odd.")
    else:
        print("Not an odd minute.")
    wait_time = random.randint(1, 60)
    time.sleep(wait_time)
```

```
word = "bottles"
for beer_num in range(99, 0, -1):
    print(beer_num, word, "of beer on the wall.")
    print(beer_num, word, "of beer.")
    print("Take one down.")
    print("Pass it around.")
    if beer_num == 1:
        print("No more bottles of beer on the wall.")
    else:
        new_num = beer_num - 1
        if new_num == 1:
            word = "bottle"
        print(new_num, word, "of beer on the wall.")
    print()
```

Мы завершили эту главу Python-версией классической «песни про пиво». И да, мы знаем: трудно работать над этим кодом, не подпевая... ☺

2 Списки

Работа с упорядоченными данными

Все программы обрабатывают данные, и программы на Python — не исключение.

На самом деле *данные повсюду*. Ведь в основном программирование — это работа с данными: *получение* данных, *обработка* данных, *интерпретация* данных. Чтобы работать с данными более эффективно, нужен какой-то контейнер, куда их можно *сложить*. Python предоставляет удобные структуры данных *широкого применения*: **списки**, **словари**, **кортежи** и **множества**. В этой главе мы бегло рассмотрим все четыре, а затем углубимся в изучение **списков** (остальные три структуры подробнее рассмотрены в следующей главе). Мы уже затрагивали эту тему ранее, поскольку все, с чем нам приходится сталкиваться при программировании на Python, так или иначе относится к работе с данными.

Числа, строки... и объекты

Работа с *единичным* значением данных в Python выглядит как обычно. Значение присваивается переменной, после чего с ним можно работать. Давайте вспомним несколько примеров из прошлой главы.

Числа

Предположим, что модуль `random` уже импортирован. Вызовем функцию `random.randint`, чтобы получить случайное число в диапазоне от 1 до 60, и присвоим его переменной `wait_time`. Поскольку это число является **целым**, переменная `wait_time` также получает целочисленный тип.

```
>>> wait_time = random.randint(1, 60)
>>> wait_time
26
```

Переменная принимает тип присваиваемого объекта.

Обратите внимание: нет необходимости явно сообщать интерпретатору, что в переменной `wait_time` будет храниться целое число. Мы просто *присвоили* целое число переменной, а о деталях интерпретатор позаботился сам (заметьте: не все языки программирования работают так же).

Строки

Если присвоить переменной строку, происходит примерно то же самое: интерпретатор сам определяет тип переменной. Опять же нет необходимости объявлять заранее, что переменная `word`, как показано в примере, содержит **строку**.

```
>>> word = "bottles"
>>> word
'bottles'
```

В Python все является объектом, и любой объект можно присвоить переменной.

Возможность *динамически* присваивать значение переменной — одна из основных особенностей переменных и типов в Python. Фактически переменной в Python можно присвоить *все что угодно*.

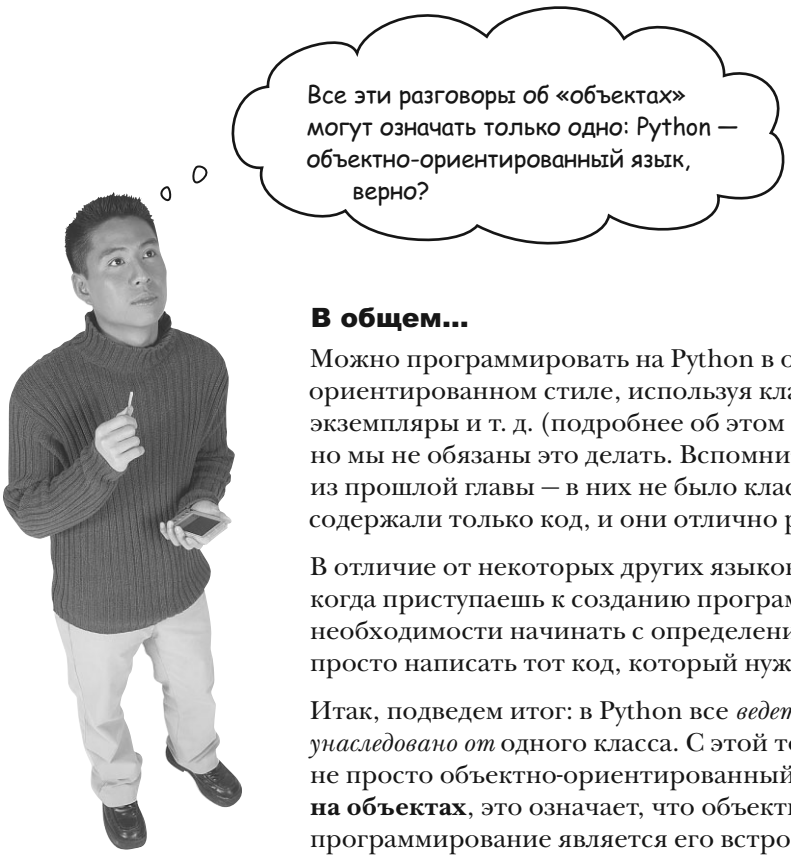
Объекты

В Python все является объектом. Это означает, что числа, строки, функции, модули — *абсолютно все* — это объекты. Следовательно, любой объект можно присвоить переменной. Некоторые интересные последствия, связанные с этим свойством, рассмотрены на следующей странице.

«Все является объектом»

В Python любой объект может быть динамически присвоен любой переменной. Тогда возникает вопрос: *что же является объектом в Python*? Ответ: **все является объектом**.

Все значения данных в Python — объекты. Например, «Don't panic!» — строка, а 42 — число. Для программистов на Python «Don't panic!» — *строковый объект*, а 42 — *числовой объект*. Как и в других языках программирования, объекты могут иметь **состояние** (атрибуты или переменные) и **поведение** (методы).



В общем...

Можно программировать на Python в объектно-ориентированном стиле, используя классы, объекты, экземпляры и т. д. (подробнее об этом рассказано далее), но мы не обязаны это делать. Вспомните хотя бы программы из прошлой главы — в них не было классов. Эти программы содержали только код, и они отлично работали.

В отличие от некоторых других языков (например, *Java*), когда приступаешь к созданию программы на Python, нет необходимости начинать с определения класса: можно просто написать тот код, который нужен.

Итак, подведем итог: в Python все *ведет себя* так, как будто *унаследовано от* одного класса. С этой точки зрения Python не просто объектно-ориентированный язык, он **основан на объектах**, это означает, что объектно-ориентированное программирование является его встроенной возможностью.

Но... что все это значит на самом деле?

Поскольку все в Python является объектом, переменной можно присвоить что угодно, и переменные могут присваиваться *чему угодно* (не важно, что это — число, строка, функция, виджет... любой объект). Но пока отложим эту тему; мы еще не раз вернемся к ней на страницах книги.

На самом деле все это не сложнее, чем просто хранить значения в переменных. Теперь обратим внимание на встроенную в Python поддержку для хранения **коллекций** значений.

Встречайте: четыре встроенные структуры данных

Python имеет **четыре** встроенные *структуры данных*, которые можно использовать для хранения *коллекций* объектов. Это **список**, **кортеж**, **словарь** и **множество**.

Обратите внимание: под «встроенными» мы подразумеваем постоянную доступность списков, кортежей, словарей и множеств в коде — их *не нужно импортировать перед использованием*: все эти структуры данных — часть языка.

На следующих страницах мы кратко рассмотрим каждую из встроенных структур данных. Возможно, вам захочется пропустить обзор, но лучше этого не делать.

Если вы думаете, что хорошо разбираетесь в **списках**, то подумайте еще раз. Дело в том, что списки в Python больше похожи на то, что мы привыкли называть *массивами* в других языках программирования, а не на *связанные списки*, которые обычно приходят на ум при слове «список». (Если вам повезло и вам никогда не встречались связанные списки, выдохните и будьте благодарны судьбе.)

Список в Python — одна из двух упорядоченных структур данных.

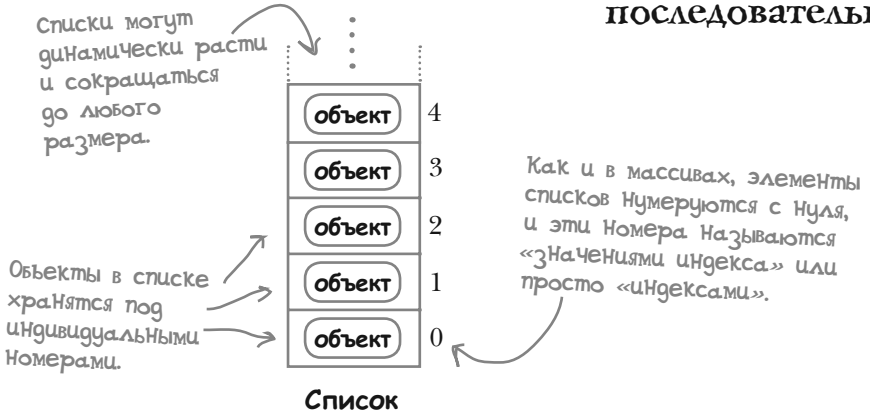
1 **Список: упорядоченная изменяемая коллекция объектов**
Списки в Python очень похожи на **массивы** в других языках программирования. Это индексированные коллекции объектов, в которых элементы нумеруются, начиная с нуля.

В отличие от массивов в других языках программирования, списки в Python являются **динамическими** структурами и количество элементов в них можно менять по требованию. Нет необходимости заранее определять размер списка.

Кроме того, списки могут быть гетерогенными, нет необходимости заранее оговаривать тип объектов, которые будут храниться в списке, — в один и тот же список можно добавлять объекты различных типов.

Списки являются **изменяемыми** структурами, это значит, что список можно изменить в любой момент, добавив, удалив или изменив объекты.

Список похож на массив — объекты хранятся в виде упорядоченной последовательности.



Упорядоченные коллекции могут быть изменяемыми и неизменяемыми

Список в Python — пример **изменяемой** структуры данных, его можно изменять во время выполнения. Списки позволяют динамически добавлять и удалять объекты по мере необходимости. Также можно изменить объект под любым индексом. Через несколько страниц мы вернемся к спискам и посвятим оставшуюся часть главы обстоятельному знакомству и приемам работы с ними.

В Python также имеется **неизменяемая** коллекция, подобная списку, которая называется **кортеж**.

2 **Кортеж: упорядоченная неизменяемая коллекция объектов**
Кортеж — это неизменяемый список. То есть после создания кортежа его нельзя изменить ни при каких обстоятельствах.

Иногда полезно рассматривать кортеж как константный список.

Большинство новичков в Python, впервые сталкиваясь с этой структурой данных, ломают голову, пытаясь понять назначение кортежей. В конце концов, в чем польза списков, которые нельзя изменять? Однако выясняется, что во многих случаях важно быть уверенным, что объект не будет изменен в клиентском коде. Мы вернемся к кортежам в главе 3 (и не раз будем встречаться с ними на протяжении книги), где подробно рассмотрим их и приемы работы с ними.

Списки и кортежи очень удобны для представления данных в упорядоченном виде (например, список пунктов назначения в туристическом маршруте, где *важен* порядок). Но иногда порядок, в котором представлены данные, *не* имеет значения. Например, при сохранении пользовательских данных (таких как *идентификатор* и *пароль*) иногда нет разницы, в каком порядке они следуют. Для таких случаев в Python существует альтернатива спискам/кортежам.

Словарь: неупорядоченная структура данных

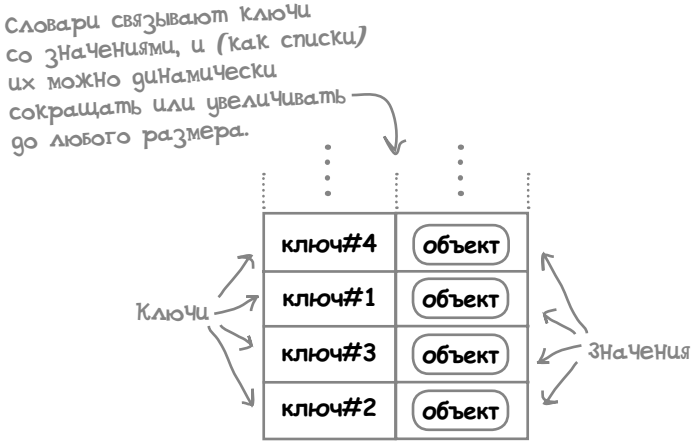
Если важен не порядок хранения данных, а их структура, то в Python для этого случая имеются две неупорядоченные структуры данных: **словари** и **множества**. Рассмотрим каждую из них и начнем со словарей.

3 **Словарь: неупорядоченное множество пар ключ/значение**
Возможно, вам уже приходилось сталкиваться со **словарями**, однако под другим названием, например ассоциативный массив, отображение, таблица символов или хеш-таблица.

Как и подобные структуры данных в других языках программирования, словари в Python позволяют хранить коллекции в виде пар ключ/значение. Каждому уникальному **ключу** ставится в соответствие **значение**, и в словаре может содержаться произвольное количество подобных пар. Значением, ассоциированным с ключом, может быть любой объект.

Словари являются неупорядоченными и изменяемыми. Их можно рассматривать как структуру данных с двумя полями. Как и списки, словари можно динамически изменять.

В словарях хранятся пары ключ/значение.



Словарь

Пользуясь словарями, помните, что нельзя полагаться на внутренний порядок следования элементов, используемый интерпретатором. Порядок, в котором пары ключ/значение добавляются в словарь, не сохраняется интерпретатором и не имеет никакого значения для Python. Такое поведение может озадачить программистов, которые впервые с ним сталкиваются, поэтому мы рассказали вам об этом сейчас, чтобы потом, в главе 3, когда мы вернемся к словарям, это не стало для вас шоком. Тем не менее, когда требуется, словари позволяют выводить данные в определенном порядке, и в главе 3 мы покажем, как это делается.

Множество: структура данных, не позволяющая дублировать объекты

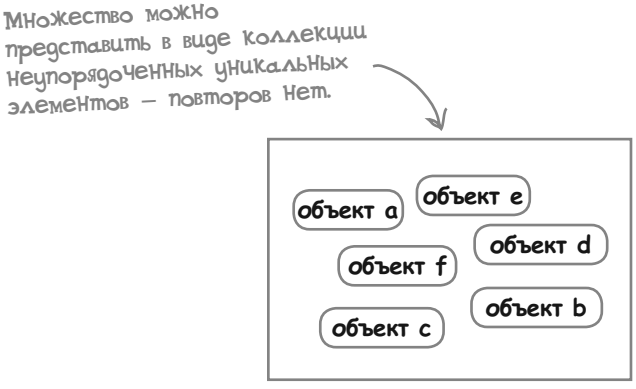
Еще одной встроенной структурой данных является **множество**. Его удобно использовать, когда нужно быстро удалить повторяющиеся значения из любой другой коллекции. Не переживайте, если при упоминании множества вы с содроганием вспоминаете школьный курс математики. Реализация множеств в Python может быть использована во многих практических случаях.

4

Множество: неупорядоченный набор неповторяющихся объектов
Множество в языке Python — это очень удобная структура данных для хранения связанных объектов, которая гарантирует, что ни один объект не повторится дважды.

Тот факт, что множества поддерживают операции объединения, пересечения и разности, является приятным дополнением (особенно для тех, кто влюблен в теорию множеств).

Множества, как списки и словари, могут расти и сокращаться динамически. Как и словари, множества являются неупорядоченными коллекциями, поэтому нельзя строить какие-либо предположения о порядке следования элементов в множестве. Более подробно множества рассмотрены в главе 3, вместе со словарями и кортежами.



Множество

Множество не позволяет дублировать объекты.

Принцип 80/20 в отношении структур данных

Эти четыре встроенные структуры данных полезны, но они не удовлетворяют всех потребностей. Хотя и перекрывают многие из них. Обычно около 80% того, что нам нужно, уже присутствует в языке, а для удовлетворения оставшихся 20% требуется выполнить дополнительную работу. Позднее мы рассмотрим, как с помощью Python удовлетворить любые индивидуальные потребности. Однако в этой и следующей главах мы рассматриваем именно встроенные структуры данных, которые позволяют покрыть 80% наших потребностей.

Оставшаяся часть главы посвящена работе с первым из четырех встроенных типов — **списком**. В следующей главе мы изучим оставшиеся три структуры — **словарь**, **множество** и **кортеж**.

Список: упорядоченная коллекция объектов

Список

Если у вас есть набор связанных объектов, которые нужно куда-то поместить, вспомните о **списке**. В списках можно хранить, например, данные об измерениях температуры за месяц.

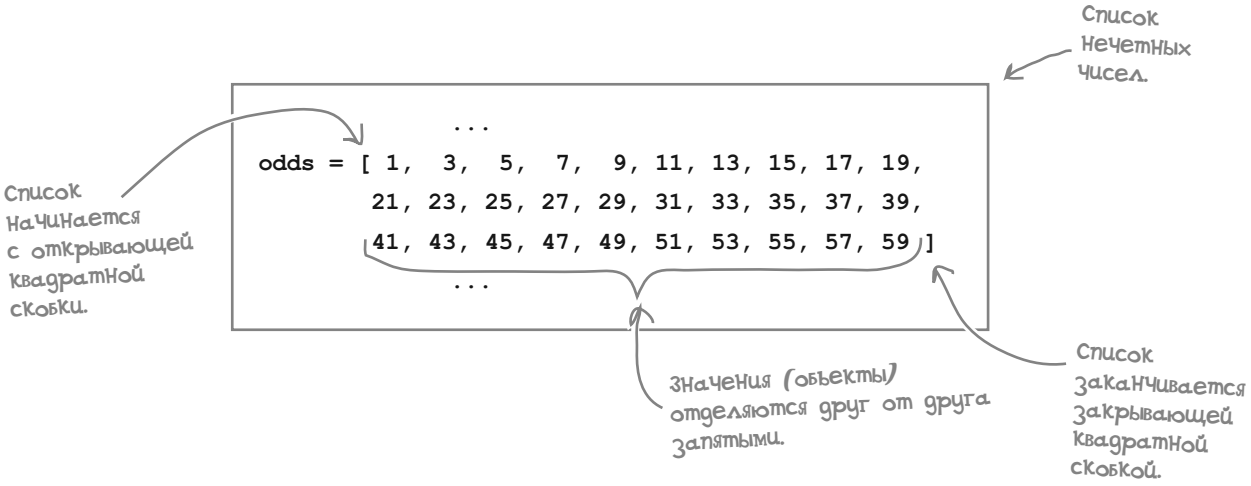
В то время как в других языках программирования массивы обычно являются гомогенными структурами данных, как, например, массив целых чисел, или массив строк, или массив измерений температуры, в Python **списки** более гибкие. Вы можете создать список *объектов*, и типы объектов могут различаться. Кроме того, списки имеют не только **гетерогенную**, но и **динамическую** природу: они могут расти и сокращаться по мере необходимости.

Прежде чем переходить к изучению приемов работы со списками, рассмотрим, как они объявляются в Python.

Как список выглядит в коде

Списки всегда заключены в **квадратные скобки**, а объекты в списке всегда отделяются друг от друга **запятыми**.

Вспомним список `odds` из предыдущей главы, в котором содержались нечетные числа от 0 до 60:



Если список создается и заполняется объектами непосредственно в коде (как показано выше), такой список называется **литеральным**; он создается *и* заполняется за один прием.

Второй способ создания и заполнения списка — постепенное добавление в него элементов в процессе выполнения программы. Далее мы увидим пример с таким способом создания списка.

Рассмотрим несколько примеров литеральных списков.

Списки можно создавать явно, перечисляя элементы, или «выращивать» программно.

Создание литеральных списков

В первом примере создается **пустой** список путем присваивания `[]` переменной `prices`.

```
prices = []
```

Имя переменной находится слева от оператора присваивания...

...а литеральный список — справа. В этом примере список пуст.

Вот список с температурами в градусах по Фаренгейту. Это список чисел с плавающей точкой.

```
temps = [ 32.0, 212.0, 0.0, 81.6, 100.0, 45.3 ]
```

Объекты (в данном случае числа с плавающей точкой) отделены друг от друга запятыми и заключены в квадратные скобки — это список.

А можно ли составить список с наиболее известными словами из области программирования? Да, пожалуйста:

```
words = [ 'hello', 'world' ]
```

Список строковых объектов.

А вот список с подробными сведениями об автомобиле. Обратите внимание, как удобно хранить в списке данные различных типов. Напомним, что список — «коллекция связанных объектов». Две строки, одно число с плавающей точкой и одно целое число — *все* это примеры объектов Python, поэтому они могут храниться в одном списке.

```
car_details = [ 'Toyota', 'RAV4', 2.2, 60807 ]
```

Список объектов различных типов.

Последние два примера еще раз подчеркивают тот факт, что в Python все является объектом. Подобно строкам, числам с плавающей точкой и целым, *списки тоже являются объектами*. Вот пример — список списков объектов.

```
everything = [ prices, temps, words, car_details ]
```

Если два последних примера пока не понятны — ничего страшного. С такими сложными объектами мы будем работать только в последующих главах.

А вот пример заполнения списка при объявлении.

Списки внутри списка.

```
odds_and_ends = [ [ 1, 2, 3], ['a', 'b', 'c' ],
                  [ 'One', 'Two', 'Three' ] ]
```

Заставим списки работать

Литеральные списки в предыдущих примерах показали, как быстро создать и заполнить список в коде. Можно просто набрать данные — и список готов.

Далее мы рассмотрим, как добавлять элементы в списки (или убирать их) во время выполнения программы. Не всегда известно, какие данные нужно хранить или сколько объектов создавать. В таких случаях в списки можно добавлять (или «генерировать») данные по мере необходимости. Пришло время разобраться, как это можно сделать.

Представим, что необходимо определить, какие гласные содержатся в слове (это буквы *a*, *e*, *i*, *o* или *u*). Можно ли использовать списки Python для решения этой задачи? Посмотрим, удастся ли нам найти решение, экспериментируя в оболочке.

Работая со списками

Используя командную строку, определим список `vowels`, а затем проверим, входит ли каждая из букв слова в этот список. Итак, определим список гласных.

Список из пяти гласных.

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
```

После определения списка `vowels` нужно выбрать слово для проверки. Создадим для этого переменную `word` и присвоим ей значение «`Milliways`».

Слово для проверки.

```
>>> word = "Milliways"
```

Для умников

В примере использовались только пять гласных букв, «*aeiou*», даже притом что буква «*y*» считается гласной и согласной одновременно.

Один объект внутри другого? Проверим с помощью «in»

Если вы помните программу из главы 1, то наверняка вспомните оператор `in`, который мы использовали, когда потребовалось определить присутствие одного объекта внутри другого. Теперь мы снова можем использовать преимущества оператора `in`:

```
>>> for letter in word:
        if letter in vowels:
            print(letter)

i
i
a
```

Берем каждую букву в слове...

...и если она содержится в списке гласных...

...отображаем эту букву на экране.

Вывод программы совпадает с гласными в слове «Milliways»

Теперь используем этот код как основу для работы со списками.

Если работаете с фрагментом кода большим, чем пара строк, используйте редактор

Список

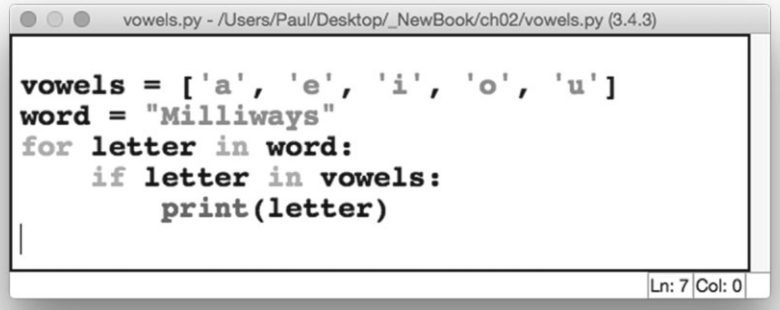
Чтобы лучше понять, как работают списки, возьмем код предыдущего примера и немного модернизируем его, чтобы каждая гласная, найденная в слове, отображалась на экране только один раз.

Для начала скопируем и вставим пример в новое окно редактирования IDLE (выберем *Файл...* →*Новый файл...* в меню IDLE). Мы собираемся внести некоторые изменения в код, поэтому удобнее работать в редакторе. Это общее правило: если код для экспериментов в командной строке занимает более пары строк, лучше работать с ним в редакторе. Сохраним эти пять строчек кода в файле `vowels.py`.

При копировании кода из командной строки в редактор **будьте внимательны** и *не* переносите группы символов >>>, иначе ваш код не будет работать (интерпретатор сообщит о синтаксической ошибке, обнаружив >>>).

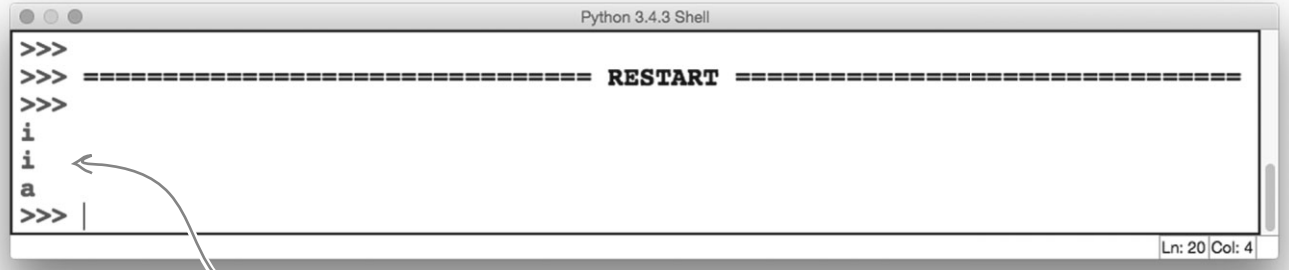
После копирования кода и сохранения файла окно редактора должно выглядеть так.

Пример использования списка, сохраненный как «vowels.py», в окне редактирования IDLE.



Не забывайте: нажимайте F5 для запуска программы.

Находясь в окне редактирования, нажмите F5 и смотрите, как IDLE выведет на передний план окно консоли и отобразит вывод программы.



Как и ожидалось, вывод программы соответствует полученному в конце прошлой страницы. И это хорошо.

Заполнение списка во время выполнения

Список

Сейчас программа *выводит* все гласные, в том числе встреченные повторно. Чтобы перечислить только единичные вхождения гласных (и избежать дублирования), нужно запомнить каждую найденную гласную до вывода ее на экран. Для этого нам необходима еще одна структура данных.

Мы не можем использовать список `vowels`, потому что он нужен, чтобы определить, является ли очередная обрабатываемая буква гласной. Нам нужен второй список, изначально пустой. Мы будем динамически заполнять его гласными, которые обнаружим в процессе работы.

Как и в предыдущей главе, поэкспериментируем в консоли, прежде чем вносить изменения в код. Чтобы создать новый, пустой список, объявим новую переменную и присвоим ей список. Назовем переменную `found`. Присвоим ей пустой список (`[]`), а затем используем встроенную функцию Python `len`, чтобы определить количество объектов в коллекции.

Встроенная функция len возвращает размер объекта.

```
>>> found = []
>>> len(found)
0
```

Пустой список...

...и интерпретатор (с помощью «len») подтверждает, что он не имеет объектов.

Списки обладают набором встроенных методов, которые можно использовать для манипулирования объектами в списке. Вызов метода оформляется с использованием *синтаксиса точечной нотации*: после имени списка добавляем точку и имя метода. В этой главе мы ознакомимся с многими методами. А пока воспользуемся методом `append`, чтобы добавить объект в конец созданного пустого списка:

```
>>> found.append('a')
>>> len(found)
1
>>> found
['a']
```

Добавим букву в существующий список с помощью метода «append».

Длина списка увеличилась.

Вывод содержимого списка в консоли подтверждает, что объект теперь часть списка.

Повторные вызовы метода `append` добавляют новые объекты в конец списка.

Списки обладают набором встроенных методов.

```
>>> found.append('e')
>>> found.append('i')
>>> found.append('o')
>>> len(found)
4
>>> found
['a', 'e', 'i' 'o']
```

Добавим еще несколько элементов.

Снова выведем содержимое списка на экран, чтобы убедиться, что все в порядке.

Рассмотрим теперь, как проверить наличие объекта в списке.

Проверка принадлежности с помощью in

Теперь мы уже знаем, как это сделать. Вспомним пример с «Millyways», несколькими страницами выше, а также код odds.py из предыдущей главы, проверяющий наличие вычисленной минуты в списке нечетных чисел odds.

Оператор in проверяет принадлежность объекта списку.

```
    ...
if right_this_minute in odds:
    print("This minute seems a little odd.")
    ...
```

Содержится или нет объект в коллекции: операторы in и not in

По аналогии с оператором in, проверяющим присутствие объекта в коллекции, комбинация операторов not in проверяет *отсутствие объекта в коллекции*.

Использование not in позволит нам добавлять элементы в список, *только* когда точно известно, что они не являются его частью.

```
>>> if 'u' not in found:
        found.append('u')

>>> found
['a', 'e', 'i' 'o', 'u']
>>>
>>> if 'u' not in found:
        found.append('u')

>>> found
['a', 'e', 'i' 'o', 'u']
```

Первый вызов «append» выполняется, потому что 'u' еще не содержится в списке «found» (поскольку, как мы видели выше, в списке только *['a', 'e', 'i', 'o']*).

Следующий вызов «append» не будет выполнен, потому что список уже содержит 'u' и добавлять этот элемент не нужно.

А не лучше ли в данном случае использовать множества? Разве множество — не лучший выбор, если нужно избежать дублирования?

Отличное замечание. Множество гораздо лучше подошло бы для этого примера.

Но мы не будем в этой главе использовать множества. А когда подойдет срок, вернемся к данному примеру. Теперь еще раз обратите внимание, как можно сгенерировать список во время работы программы с помощью метода append.

Пора внести изменения в код

Теперь, после знакомства с `not in` и `append`, можно внести изменения в код. Ниже еще раз приводится оригинальный код из файла `vowels.py`:

Оригинальный код в «vowels.py».

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

Этот фрагмент выводит гласные, имеющиеся в «word», в порядке их нахождения.

Сохраните код в файле с именем `vowels2.py`, чтобы вносить изменения в новую версию программы, сохранив оригинал.

Теперь мы должны создать пустой список `found` и добавить код, заполняющий список `found` во время выполнения программы. Так как мы не будем выводить гласные в процессе поиска, необходим еще один цикл `for`, чтобы обработать буквы в списке `found`, и второй цикл `for` должен выполняться *после* первого (обратите внимание, как *согласуются* отступы в обоих циклах). Новый код, который нужно добавить, выделен на рисунке.

Это файл «vowels2.py».

Начните отсюда, добавив создание пустого списка.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Добавьте код, определяющий необходимость включения буквы в список найденных гласных.

Когда заканчивается первый цикл «for», начинает выполняться второй, который отображает найденные в слове гласные.

Добавим финальный штрих в этот код, изменив строку, которая присваивает переменной `word` значение «Milliways», чтобы сделать программу более *гибкой* и *интерактивной*.

Изменим строку

```
word = "Milliways"
```

вот так:

```
word = input("Provide a word to search for vowels: ")
```

Она сообщает интерпретатору о необходимости запросить у пользователя строку для поиска гласных. Функция `input` — еще одна встроенная функция Python.

Сделай это!

Внесите изменения, предложенные слева, и сохраните код в файле с именем `vowels3.py`.

Пробная поездка

Внеся изменения, которые мы обсудили в конце прошлой страницы, и сохранив их в файле с именем `vowels3.py`, давайте протестируем эту программу и запустим ее в среде IDLE. Напоминаем: чтобы запустить программу, необходимо сначала вернуться в окно редактора IDLE, а затем нажать клавишу F5.

Это наша версия «vowels3.py» с функцией «input» в окне редактора.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

А это результаты пробных запусков программы.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Milliways
i
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Hitch-hiker
i
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Sky
>>>
```

Ln: 21 Col: 4

Вывод подтверждает, что программа работает, как ожидается, и *работает правильно*, даже если слово не содержит гласных совсем. Получился ли у вас такой же результат?

Удаление объектов из списка

Списки в Python похожи на массивы в других языках программирования.

Способность списков расти динамически, когда требуется больше пространства (благодаря методу `append`), является большим преимуществом. Как и во многих других случаях, интерпретатор Python сам позаботится обо всем, что нужно. Если списку нужно больше места, интерпретатор *выделит* еще память. Когда список сокращается, интерпретатор динамически *освободит* память, которая уже не нужна.

Есть и другие методы для работы со списками. На следующих четырех страницах мы рассмотрим четыре наиболее полезных метода: `remove`, `pop`, `extend` и `insert`:

1. **remove: принимает значение объекта в единственном аргументе**
Метод `remove` удаляет первое вхождение указанного значения из списка. Если значение обнаружится в списке, объект, содержащий это значение, будет удален из списка (и список сократится на один элемент). Если значение *не* обнаружится в списке, интерпретатор *возбудит ошибку* (это мы обсудим далее):

```
>>> nums = [1, 2, 3, 4]
>>> nums
[1, 2, 3, 4]
```

Это список «nums» до вызова метода «remove».

Это *не* индекс, это значение, которое нужно удалить.

```
>>> nums.remove(3)
>>> nums
[1, 2, 4]
```

После вызова метода «remove» объект со значением 3 пропал.

Извлечение объектов из списка

Метод `remove` удобен, если заранее известно значение объекта для удаления. Но часто требуется удалить элемент по определенному индексу. Для таких случаев в Python есть метод `pop`:

2

pop: принимает необязательный аргумент с индексом
Метод `pop` удаляет из списка *и возвращает* объект по индексу. Если метод `pop` вызвать без индекса, будет удален и возвращен последний объект в списке. Если указан индекс, будет удален и возвращен объект согласно индексу. Если список пуст или метод `pop` вызван с несуществующим индексом, интерпретатор *возбудит ошибку* (об этом мы поговорим позже).

Объект, возвращаемый методом `pop`, можно присвоить переменной, если есть такая необходимость, и тем самым сохранить его. Однако если извлеченный объект не присваивается переменной, память очищается и объект перестает существовать.

До вызова «pop» список содержал 3 элемента.

Мы не сообщили «pop», какой элемент удалить, поэтому он удалит последний элемент списка.

```
>>> nums.pop()
4
>>> nums
[1, 2]
```

Метод «pop» возвращает удаленный объект.

После вызова метода «pop» список сократился.

Как и в предыдущем случае, метод «pop» возвращает удаленный объект, который утилизируется интерпретатором.

Это значение индекса. Ноль соответствует первому элементу списка (номер 1).

```
>>> nums.pop(0)
1
```

Список «nums» сократился до одного элемента.

```
>>> nums
[2]
```

А теперь в списке остался единственный элемент.

Добавление элементов в список

Теперь вы знаете, что добавить единственный объект в список можно с помощью метода `append`. Однако имеются и другие методы для динамического добавления данных в список:

3 **extend: принимает список объектов в единственном аргументе**
Метод `extend` принимает еще один список и добавляет все его элементы в существующий список. Метод очень удобно использовать для объединения двух списков.

Вот так в настоящий момент выглядит список «nums»: всего один элемент.

```
>>> nums.extend([3, 4])
[2, 3, 4]
```

Список элементов для добавления к существующему списку.

Мы расширили список «nums», добавив в конец каждый элемент из списка-аргумента.

```
>>> nums.extend([])
[2, 3, 4]
```

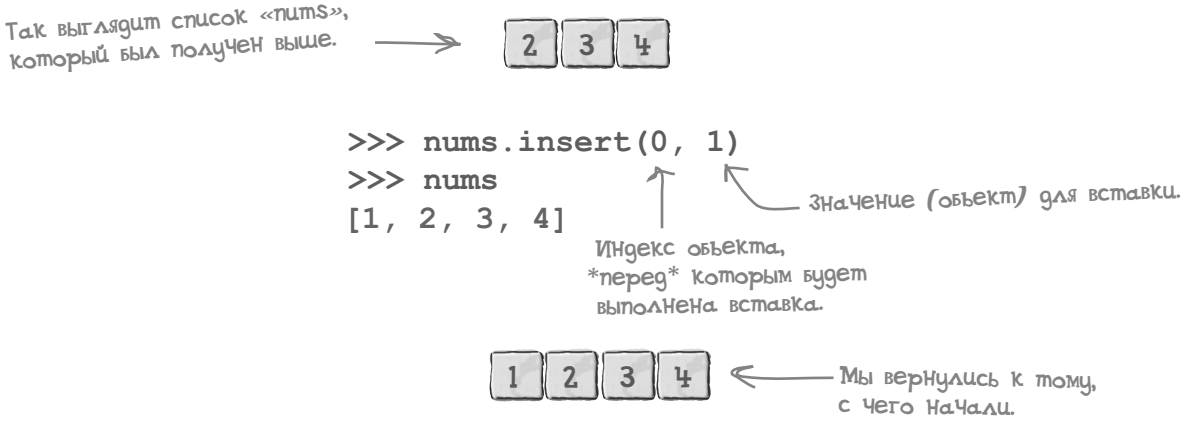
Использование пустого списка возможно, но не разумно (поскольку в этом случае мы не добавляем элементы в конец списка). Если вместо этого вызвать метод «append([])», в конец существующего списка будет добавлен пустой список. Но в этом примере вызов «extend([])» не выполняет никакой работы.

Поскольку пустой список, использованный для расширения «nums», не содержит никаких объектов, ничего не изменилось.

Вставка элементов в список

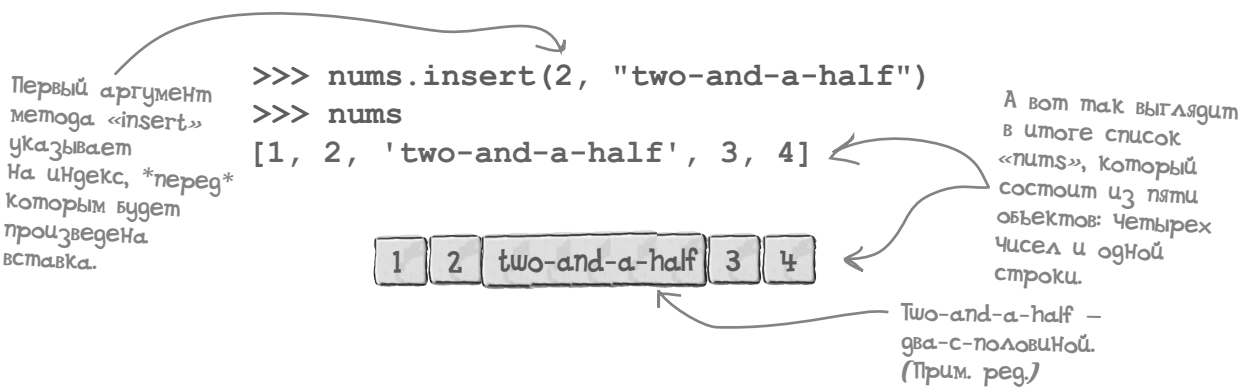
Методы `append` и `extend` очень полезны, но они имеют одно ограничение — объекты добавляются только в конец списка (справа). Иногда необходимо добавить элементы и в начало (слева). Для таких случаев существует метод `insert`.

4 **insert: принимает в качестве аргументов значение индекса и объект для вставки**
Метод `insert` вставляет объект в существующий список *перед* индексом, указанным в первом аргументе. Это позволяет вставить объект в начало списка или в любое другое место, но только не в конец списка, как это делает метод `append`.



После всех удалений, извлечений, расширений и вставок в итоге мы получили тот же список, с которого начали несколькими страницами выше: `[1, 2, 3, 4]`.

Обратите внимание, что `insert` можно использовать для добавления объекта по любому индексу в имеющемся списке. В примере выше мы решили добавить объект (число `1`) в начало списка, однако мы можем вставить элемент по любому индексу *в* списке. В заключительном примере рассмотрим, как — просто ради забавы — добавить строку в середину списка «`nums`». Для этого передадим в первом аргументе методу `insert` индекс `2`.



А теперь поупражняемся в использовании этих методов.

Как использовать квадратные скобки

Что-то я запуталась. Вы говорите, что списки похожи на массивы в других языках программирования, но так ничего и не сказали о моих любимых квадратных скобках, которые я привыкла использовать для обращения к элементам массива в других моих любимых языках программирования. Что скажете?

Не беспокойтесь, мы еще это рассмотрим.

Привычная запись с квадратными скобками, известная и излюбленная при работе с массивами в других языках программирования, действительно может применяться при работе со списками Python. Однако прежде чем обсудить это, давайте еще потратим немного времени и потренируемся применять уже изученные нами методы.

Это не Глупые вопросы

В: **Как найти больше информации по этим и другим методам работы со списками?**

О: Нужно обратиться к справке. Наберите в командной строке >>> команду ***help(list)***, чтобы получить документацию Python по спискам (содержащую несколько страниц описания) или наберите ***help(list.append)***, чтобы получить документацию только по методу `append`. Замените `append` любым другим методом, который вас интересует.

Заточите карандаш

Время для испытаний.

Прежде чем продолжить, наберите в новом окне редактора семь строк кода, напечатанные в конце этой страницы. Сохраните в файле с именем `panic.py` и выполните его, нажав F5.

Изучите сообщения на экране. Обратите внимание, как строка (в переменной `phrase`) преобразуется в список (в переменной `plist`), а затем на экране отображаются обе переменные: `plist` и `phrase`.

Следующие три строки кода принимают `plist` и трансформируют его обратно в строку (в переменной `new_phrase`), а затем выводят и список, и полученную строку на экран.

Ваша задача: *трансформировать* строку «`Don't panic!`» в строку «`on tap`», используя методы списков, рассмотренные выше. (В выборе этих двух строк нет никакого скрытого смысла: просто в строке «on tap» присутствуют символы, которые имеются в строке «`Don't panic!`»). В данный момент программа `panic.py` *дважды* выводит «`Don't panic!`».

Подсказка: используйте цикл `for` для выполнения какой-либо операции несколько раз.

Мы начали со строки.

Превратили строку в список.

Мы вывели на экран строку и список.

Вставьте здесь ваш код для работы со списком.

Вывели изменённый список и новую строку.

Здесь список превращается обратно в строку.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
.....
.....
.....
.....
.....
.....
.....
.....
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Заточите карандаш
Решение

Это было время испытаний.

Прежде чем продолжить, нужно было набрать в новом окне редактора семь строк кода с предыдущей страницы и сохранить их в файле с именем `panic.py`, а затем выполнить (нажав F5).

Задача была в том, чтобы *трансформировать* строку «`Don't panic!`» в строку «`on tap`», используя только методы списков, рассмотренные выше. До начала работы программа `panic.py` дважды выводила строку «`Don't panic!`».

Новая строка («`on tap`») должна быть сохранена в переменной `new_phrase`.

Код обработки списка нужно было добавить сюда. Мы приведем свой пример кода — не переживайте, если он отличается от вашего. Есть несколько способов выполнить задание.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

```
for i in range(4):
    plist.pop()
```

Этот маленький цикл извлекает последние четыре объекта из «plist», удаляя из строки «nic!»

Извлечем первую букву списка ('D').

```
plist.pop(0)
```

```
plist.remove(" ' ")
```

Находим и удаляем апостроф.

```
plist.extend([plist.pop(), plist.pop()])
```

Поменяем местами два объекта в конце списка, предварительно вынув каждый объект из списка, а затем использовав их для расширения. Возможно, вам придется поразмыслить немного над этой строкой кода. Ключ: *сначала* элементы извлекаются из списка (начиная с конца), а затем добавляются.

```
plist.insert(2, plist.pop(3))
```

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Эта строка кода извлекает пробел из списка и вставляет его обратно в список, но перед элементом с индексом 2. Как и в предыдущей строке кода, сначала происходит извлечение пробела, а затем вставка. И запомните: пробелы тоже являются символами.

На следующих страницах рассмотрим, что происходит в коде, более подробно.

Что произошло в plist?

Остановимся на минутку и задумаемся, что же на самом деле произошло в `plist` во время выполнения `panic.py`.

Слева на этой (и на следующей) странице представлен код из `panic.py`, который, как и все программы на Python, выполняется сверху вниз. Справа находятся графическое представление `plist` и некоторые замечания к происходящему. Обратите внимание, как `plist` динамически сокращается и растет во время выполнения кода:

Список

Код

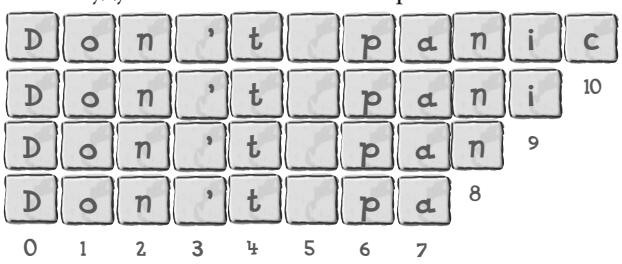
```
phrase = "Don't panic!"

plist = list(phrase)

print(phrase)
print(plist)

for i in range(4):
        plist.pop()

plist.pop(0)

plist.remove("'")
```

Здесь вызывается функция print для вывода на экран текущего состояния переменных (до начала манипуляций с ними).

Состояние `plist`

В данной точке программы `plist` еще не существует. Вторая строка кода *трансформирует* строку `phrase` в новый список, который присваивается переменной `plist`:

В каждой итерации цикла `for` список `plist` сокращается на один элемент, пока из него не будут извлечены все четыре объекта:



Цикл завершается, и в `plist` осталось 8 элементов. Теперь избавимся от некоторых ненужных объектов. Еще один вызов `pop` удалит из списка первый элемент (с индексом `0`).

После удаления заглавной буквы `D` из начала списка с помощью метода `remove` удаляется апостроф.

Что произошло в plist, продолжение

Мы остановились на минутку, чтобы понять, что же на самом деле произошло в `plist` во время выполнения `panic.py`.

После выполнения кода, рассмотренного выше, в нашем списке осталось 6 элементов, символы: `o`, `n`, `t`, пробел, `p`, `a`. Продолжим выполнение кода:

Код

```
plist.extend([plist.pop(), plist.pop()])
```

Состояние `plist`

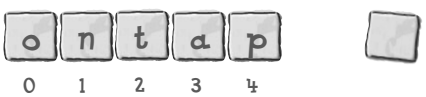
Так выглядит `plist` после выполнения кода выше.

Следующая строка содержит вызовы **трех** методов: два вызова `pop` и один вызов `extend`. Сначала идут вызовы `pop` (слева направо):

Во время вызова метода `extend` извлеченные объекты добавляются в конец списка `plist`. Иногда удобно использовать один вызов метода `extend` вместо нескольких вызовов метода `append`:

Осталось (в списке `plist`) переместить букву `t` в ячейку с индексом `2`, а пробел — в ячейку с индексом `3`. Следующая строка содержит вызовы двух методов. Первый вызов `pop` извлекает пробел:

```
plist.insert(2, plist.pop(3))
```

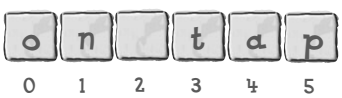


Превращает «plist» обратно в строку.

```
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Эти вызовы «print» выводят на экран содержимое переменных (после наших манипуляций с ними).

А затем вызов `insert` помещает пробел в нужное место (перед элементом с индексом `2`):



Тадам!

Списки: что мы узнали

Вот мы и прочитали 20 страниц. Теперь сделаем перерыв и вспомним, что же мы узнали о списках.

КОНТРОЛЬНЫЙ СПИСОК

- Списки удобны для хранения коллекции связанных объектов. Если у вас есть набор похожих элементов, которые хотелось бы воспринимать как один общий объект, их можно положить в список.
- Списки похожи на массивы в других языках программирования. Однако в отличие от массивов в других языках (которые обычно имеют фиксированный размер), списки в Python могут расти или сокращаться динамически по мере необходимости.
- В коде список объектов заключен в квадратные скобки, внутри списка объекты отделяются друг от друга запятыми.
- Пустой список представляется так: `[]`.
- Быстро проверить присутствие элемента в списке можно с помощью оператора `in`.
- Рост списков во время выполнения программы может происходить в результате использования методов `append`, `extend` и `insert`.
- Сокращение списков во время выполнения программы может происходить в результате вызова методов `remove` и `pop`.

Да. Внимание нужно всегда.

Работа со списками в Python организована очень удобно, но все равно важно убедиться, что интерпретатор выполняет именно то, что вы задумали.

В качестве примера рассмотрим копирование одного списка в другой. Мы копируем список или объекты в списке? В зависимости от ответа и желаемого результата, интерпретатор должен вести себя по-разному. Переверните страницу, чтобы узнать, что мы имеем в виду.

Выглядит как копия, но не копия

Когда требуется скопировать один список в другой, возникает соблазн использовать оператор присваивания:

```
>>> first = [1, 2, 3, 4, 5]
>>> first
[1, 2, 3, 4, 5]
>>> second = first
>>> second
[1, 2, 3, 4, 5]
```

Создать новый список (и положить в него пять числовых объектов).

Пять элементов в списке «first».

«Скопировать» существующий список в новый, который назовем «second».

Пять элементов в списке «second».

Пока все нормально. Код, похоже, работает, потому что пять объектов из `first` скопированы в `second`:

Но действительно ли это так? Посмотрим, что получится, если добавить несколько элементов в список `second` при помощи метода `append`. Казалось бы, совершенно нормальное действие, но оно приводит к проблеме:

```
>>> second.append(6)
>>> second
[1, 2, 3, 4, 5, 6]
```

Кажется, что все нормально. Но это не так.

Опять вроде бы все нормально — но здесь **ошибка**. Посмотрите, что произойдет, если вывести на экран содержимое списка `first`, — новый объект также добавится в список `first`!

```
>>> first
[1, 2, 3, 4, 5, 6]
```

Упс! Новый объект добавился также в список «first»!

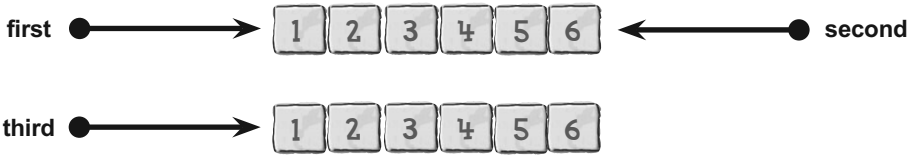
Проблема в том, что обе переменные — и `first`, и `second` — ссылаются на одни и те же данные. Если поменять один список, другой тоже изменится. И это нехорошо.

Как скопировать структуру данных

Если оператор присваивания не копирует один список в другой, тогда что он делает? Он записывает в переменные `first` и `second` **ссылку** на *один и тот же* список.

Чтобы решить эту проблему, нужно использовать метод `copy`, который выполняет именно то, что нужно. Посмотрим, как работает `copy`:

```
>>> third = second.copy()
>>> third
[1, 2, 3, 4, 5, 6]
```



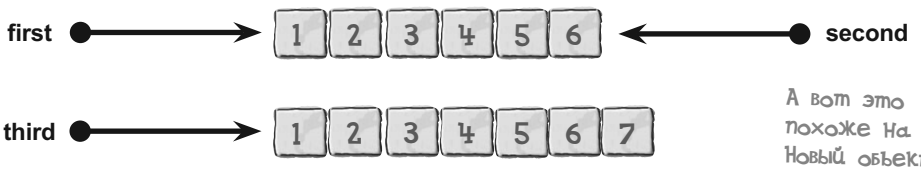
Создав список `third` (благодаря методу `copy`), добавим в его конец объект и посмотрим, что получится:

Список «third» вырос на один объект.

```
>>> third.append(7)
>>> third
[1, 2, 3, 4, 5, 6, 7]
>>> second
[1, 2, 3, 4, 5, 6]
```

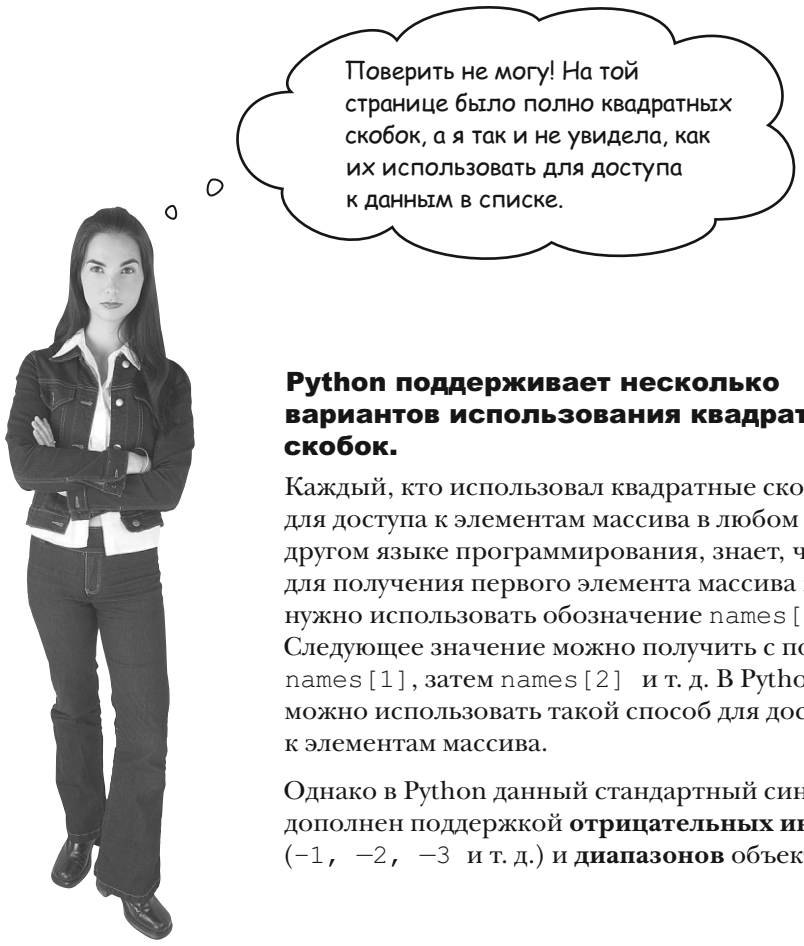
Намного лучше. Существующий список не изменился.

Не используйте оператор присваивания, чтобы скопировать список. Используйте метод «copy».



А вот это похоже на правду. Новый объект добавлен только в список «third», но не в остальные два («first» и «second»).

Квадратные скобки повсюду



Python поддерживает несколько вариантов использования квадратных скобок.

Каждый, кто использовал квадратные скобки для доступа к элементам массива в любом другом языке программирования, знает, что для получения первого элемента массива `names` нужно использовать обозначение `names[0]`. Следующее значение можно получить с помощью `names[1]`, затем `names[2]` и т. д. В Python тоже можно использовать такой способ для доступа к элементам массива.

Однако в Python данный стандартный синтаксис дополнен поддержкой **отрицательных индексов** (`-1, -2, -3` и т. д.) и **диапазонов** объектов.

Списки: добавим кое-что к изученному

Прежде чем рассмотреть, как Python расширяет нотацию с квадратными скобками, дополним наш контрольный список.

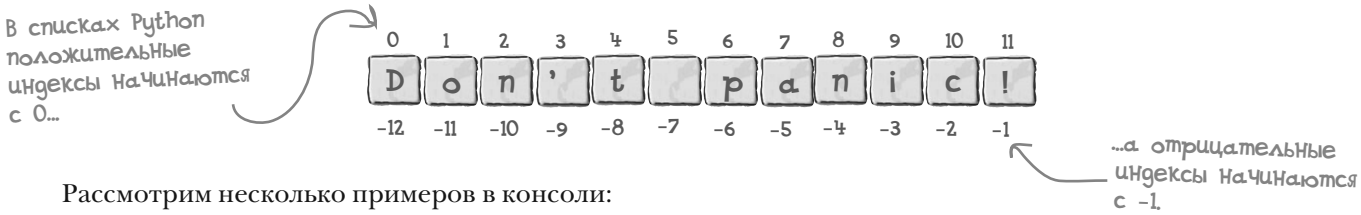
КОНТРОЛЬНЫЙ СПИСОК

- Будьте внимательны при копировании одного списка в другой. Если вы хотите, чтобы еще одна переменная ссылалась на существующий список, используйте для ее инициализации оператор присваивания (=). Если вы хотите скопировать объекты из существующего списка и инициализировать ими новый список, используйте метод `copy`.

Списки расширяют нотацию с квадратными скобками

Наши разговоры о том, что списки в Python похожи на массивы в других языках программирования, — это не пустые разговоры. Как и в других языках, списки в Python нумеруются с нуля и поддерживают возможность **использования квадратных скобок** для доступа к их элементам.

В отличие от большинства других языков Python позволяет обращаться к элементам списка либо относительно начала, либо относительно конца: положительные индексы ведут счет слева направо, а отрицательные — справа налево.



Рассмотрим несколько примеров в консоли:

```
>>> saying = "Don't panic!"
>>> letters = list(saying)
>>> letters
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
>>> letters[0]
'D'
>>> letters[3]
"'"
>>> letters[6]
'p'
>>> letters[-1]
'!'
>>> letters[-3]
'i'
>>> letters[-6]
'p'
```

Создадим список букв.

Положительные значения индексов отсчитываются слева направо.

А отрицательные значения индексов отсчитываются справа налево.

Очень просто получить первый и последний объекты в любом списке.

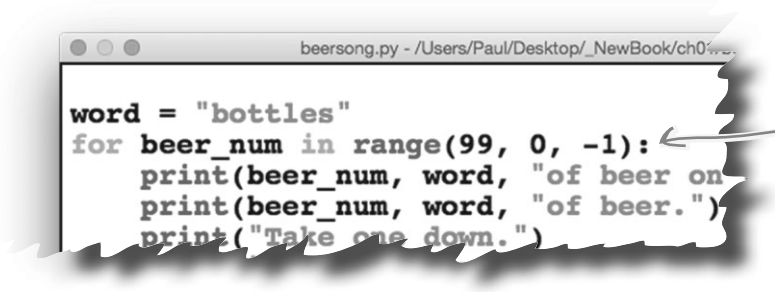
```
>>> first = letters[0]
>>> last = letters[-1]
>>> first
'D'
>>> last
'!'
```

Поскольку списки могут увеличиваться и сокращаться во время выполнения, использование отрицательных индексов часто бывает удобным. Так, по индексу `-1` всегда можно получить последний объект в списке, при этом *не важен сам размер списка*, а по индексу `0` всегда находится первый объект.

Расширенная поддержка синтаксиса квадратных скобок в Python не ограничивается поддержкой отрицательных индексов. В квадратных скобках можно также указывать диапазоны (**начало**, **конец** и **шаг**).

Со списками можно использовать диапазоны

Впервые с **началом**, **окончанием** и **шагом** мы встретились в главе 1 при обсуждении функции `range` с тремя аргументами:



В вызов «range» передаются три аргумента – начало, конец, шаг.

Напомним, что **начало**, **конец** и **шаг** определяют диапазоны значений (а теперь свяжем это со списками):

- **НАЧАЛЬНОЕ значение позволяет контролировать, ГДЕ начинается диапазон.**
  В случае со списками **начало** означает начальный индекс.

- **КОНЕЧНОЕ значение позволяет определить, ГДЕ диапазон заканчивается.**
  При работе со списками – это **индекс**, где диапазон заканчивается, но данный индекс в диапазон **не включается**.

- **Значение ШАГА определяет, КАК генерируется диапазон.**
  При работе со списками **шаг** определяет *расстояние*, преодолеваемое за один раз во время движения через список.

Начало, конец и шаг можно поместить внутрь квадратных скобок

При использовании со списками начало, конец и шаг указываются внутри квадратных скобок и отделяются друг от друга двоеточием (`:`)

```
letters[start:stop:step]
```

Расширенный синтаксис квадратных скобок позволяет указывать начало, конец и шаг.

Это может показаться странным, но все три значения являются *необязательными*:

Если опущено **начальное** значение, используется значение по умолчанию `0`.

Если опущено **конечное** значение, используется максимальный индекс в списке.

Если опущен **шаг**, по умолчанию используется `1`.

Срезы в действии

Для существующего списка `letters`, который описан несколькими страницами ранее, также можно использовать **начало**, **конец** и **шаг** различными способами.

Рассмотрим примеры:

```
>>> letters
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
```

Все буквы.

```
>>> letters[0:10:3]
['D', "'", 'p', 'i']
```

Каждая третья буква до (но не включая) индекса 10.

```
>>> letters[3:]
["'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
```

Пропустить первые три буквы и вернуть все остальные.

```
>>> letters[:10]
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i']
```

Все буквы до (но не включая) индекса 10.

```
>>> letters[::2]
['D', 'n', 't', 'p', 'n', 'c']
```

Каждая вторая буква.

Использование *синтаксиса срезов* с применением начала, конца и шага при работе со списками — очень мощный (не говоря уж удобный) прием, и мы советуем вам потратить немного времени, чтобы понять, как работают эти примеры. Запустите их в своей консоли, а затем поэкспериментируйте самостоятельно.

Это не Глупые вопросы

В: Я заметил, что некоторые символы на этой странице заключены в одиночные, а другие — в двойные кавычки. Есть ли какой-нибудь стандарт, которому нужно следовать?

О: Нет, стандарта нет, и Python позволяет использовать либо одиночные, либо двойные кавычки для выделения строк любой длины, включая односимвольные строки (как те, которые встретились на этой странице; технически, это односимвольные строки, а не буквы). Большинство программистов на Python используют одиночные кавычки для выделения строк (но это пожелание, а не правило). Если в строке есть одиночные кавычки, можно использовать двойные кавычки и избежать применения обратной косой черты (\), потому что обычно проще прочитать «`'`», чем `'\''`. На следующих двух страницах вы увидите еще несколько примеров использования кавычек обоих видов.

Начало и конец диапазона в списках

Обязательно опробуйте примеры на этой (и следующей) странице в консоли, чтобы убедиться, что все работает правильно.

Начнем с превращения строки в список букв.

Превратим строку в список и выведем список на экран.

```
>>> book = "The Hitchhiker's Guide to the Galaxy"
>>> booklist = list(book)
>>> booklist
['T', 'h', 'e', ' ', 'H', 'i', 't', 'c', 'h', 'h', 'i', 'k',
'e', 'r', "'", 's', ' ', 'G', 'u', 'i', 'd', 'e', ' ', 't',
'o', ' ', 't', 'h', 'e', ' ', 'G', 'a', 'l', 'a', 'x', 'y']
```

Обратите внимание, что в строке есть символ одиночной кавычки. Python понимает это и заключает одиночную кавычку в двойные.

Затем используем новый список `booklist` для выбора диапазона букв.

```
>>> booklist[0:3]
['T', 'h', 'e']
```

Выберем первые три объекта (буквы) из списка.

```
>>> ''.join(booklist[0:3])
'The'

>>> ''.join(booklist[-6:])
'Galaxy'
```

Превратим выбранный диапазон в строку (как это сделать, вы узнали в примере «panic.py»). Второй пример выбирает последние шесть объектов из списка.

Обязательно уделите время изучению этой страницы (и следующей), пока не поймете, как работает каждый из примеров, обязательно запустите каждый из них в IDLE.

Обратите внимание, как в последнем примере интерпретатор использует значения по умолчанию для определения границ диапазона и шага.

Шаг диапазона в списках

Вот еще два примера, как можно организовать **обход** списков.

Первый пример выбирает все буквы с конца (то есть *в обратном направлении*), а второй — каждую вторую букву в списке. Обратите внимание, как это организовано с помощью **шага**.

```
>>> backwards = booklist[::-1]
>>> ''.join(backwards)
"yxalaG eht ot ediuG s'rekihhctiH ehT"
```

Похоже на абракадабру, правда? Но это всего лишь перевернутая входная строка.

```
>>> every_other = booklist[::2]
>>> ''.join(every_other)
"TeHthie' ud oteGlx"
```

И этот результат похож на тарабарщину! Но список «every_other» составлен из каждого второго элемента списка (буквы), начиная с первого и заканчивая последним. Обратите внимание: начало и конец выбраны по умолчанию.

Эти примеры подтверждают возможность произвольного выбора начала и конца диапазона выбираемых элементов в списке. Возвращаемые в таком случае данные являются **срезом**. Их можно рассматривать как *фрагмент* существующего списка.

Два примера ниже выбирают из списка `booklist` слово `'Hitchhiker'`. Первый из них выбирает слово в прямом, а второй — в обратном направлении:

Срез — фрагмент списка.

```
>>> ''.join(booklist[4:14])
'Hitchhiker'
```

Вырезать слово «Hitchhiker».

```
>>> ''.join(booklist[13:3:-1])
'rekihhctiH'
```

Вырезать слово «Hitchhiker», но в обратном направлении.

Срезы повсюду

Синтаксис срезов можно применять не только к спискам. Фактически в Python можно получать срезы из любых последовательностей, поддерживающих нотацию **`[start:stop:step]`**.

Работаем со срезами в списке

Срезы в Python — полезное расширение нотации с квадратными скобками, их можно использовать со многими языковыми конструкциями. На страницах книги вам встретятся многочисленные подтверждения этому.

Теперь посмотрим, как работает нотация с квадратными скобками. Возьмем программу `panic.py` и преобразуем ее с помощью квадратных скобок и срезов, чтобы решить эту задачу проще, чем с применением методов списков.

Прежде чем приступить к работе, вспомним, что делает `panic.py`.

Преобразование «Don't panic!» в «on tap»

Этот код преобразует одну строку в другую, используя для этого существующий список и методы работы со списками. Начав со строки «`Don't panic!`», код преобразует ее в строку «`on tap`»:

Это «panic.py».

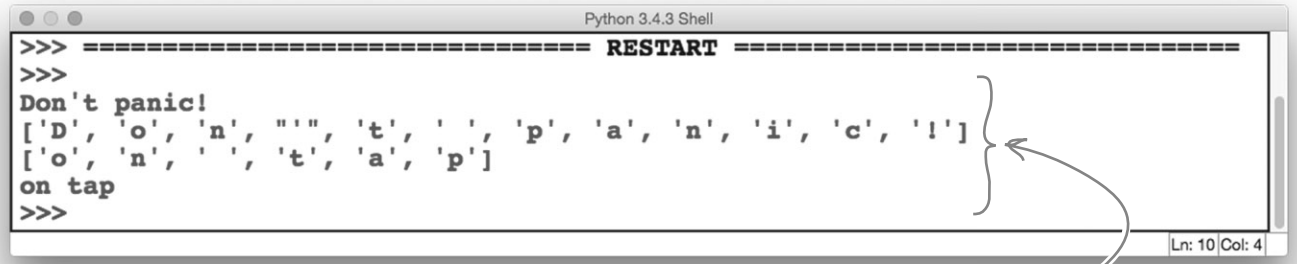
Вывести на экран начальное содержимое строки и списка.

Используя методы списков, преобразовать список объектов.

Вывести на экран результат — строку и список.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Вот результат работы программы в среде IDLE.



Строка «Don't panic!» преобразована в «on tap» с помощью методов списков.

Работаем со срезами в списке, продолжение

Пришло время поработать. Ниже показан код `panic.py`, в котором выделен фрагмент, подлежащий изменению.

Это строки кода, которые необходимо изменить.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))
new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Заточите карандаш

В упражнении нужно заменить выделенные строки кода новыми, с использованием квадратных скобок. Обратите внимание, что при необходимости можно использовать методы списков. Как и прежде, нужно преобразовать строку «`Don't panic!`» в «`on tap`». Внесите необходимые изменения и сохраните программу с именем `panic2.py`.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
.....

.....

.....

.....

.....
print(plist)
print(new_phrase)
```

Заточите карандаш
Решение

В упражнении нужно заменить выделенные строки кода новыми, с использованием квадратных скобок. Обратите внимание, что при необходимости можно использовать методы списков. Как и прежде, нужно преобразовать строку «Don't panic!» в «on tap». Внесите необходимые изменения и сохраните программу с именем panic2.py.

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)
```

Мы начнем с вырезания «on» из «plist»...

```
new_phrase = ''.join(plist[1:3])
```

```
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```

...а затем выберем каждый нужный символ отдельно: пробел, «t», «a» и «p».

```
print(plist)
print(new_phrase)
```

Хороший вопрос.

Некоторые программисты сравнят код panic2.py с кодом в panic.py и решат, что две строки кода всегда лучше, чем семь, особенно если программы делают одно и то же. Программы можно сравнивать по длине, но это не всегда полезно.

Чтобы понять, что мы имеем в виду, взглянем на результаты работы программ.

Пробная поездка

Используя IDLE, откроем `panic.py` и `panic2.py` в разных окнах редактора. Выберем окно с `panic.py` и нажмем F5. Затем выберем окно `panic2.py` и тоже нажмем F5. Сравним результаты работы обеих программ.

«panic.py».

panic.py - /Users/Paul/Desktop/_NewBook/ch02/panic.py (3.4.3)

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Ln: 17 Col: 0

«panic2.py».

panic2.py

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ================================ RESTART ================================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

Вывод программы «panic.py».

Вывод программы «panic2.py».

Обратите внимание на различия в выводе.

Какая программа лучше? Зависит от...

Мы выполнили обе программы — `panic.py` и `panic2.py` — в IDLE, чтобы было проще определить, какая из них «лучше».

Посмотрите на вторую снизу строку в выводе обеих программ.

Это вывод программы «panic.py»...

```
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['o', 'n', ' ', 't', 'a', 'p']
on tap
>>> ========================== RESTART ==========================
>>>
Don't panic!
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
['D', 'o', 'n', "'", 't', ' ', 'p', 'a', 'n', 'i', 'c', '!']
on tap
>>>
```

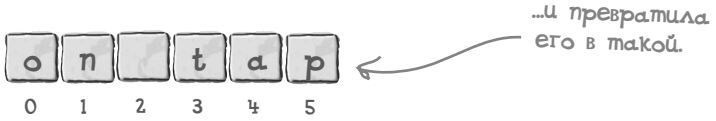
...а это вывод программы «panic2.py».

Хотя обе программы завершаются после вывода строки «`on tap`» (начав со строки «`Don't panic!`»), программа `panic2.py` не вносит изменения в список `plist`, а `panic.py` — вносит.

Стоит задержаться на минутку, чтобы обдумать это.

Вспомните обсуждение в разделе *«Что произошло в plist?»*, выше. В ходе той дискуссии мы по шагам рассмотрели, как этот список:

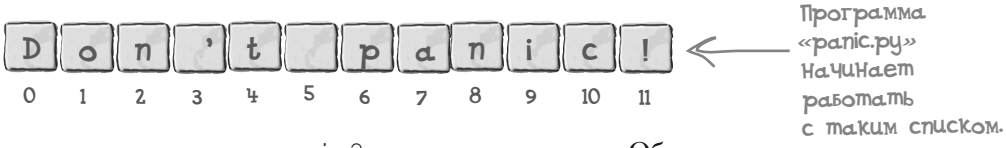
превращается в более короткий:



Все эти манипуляции с использованием `pop`, `remove`, `extend` и `insert` изменяют исходный список, и это правильно, потому что все эти методы созданы именно для внесения изменений в списки. А что происходит в `panic2.py`?

Срезы не изменяют список

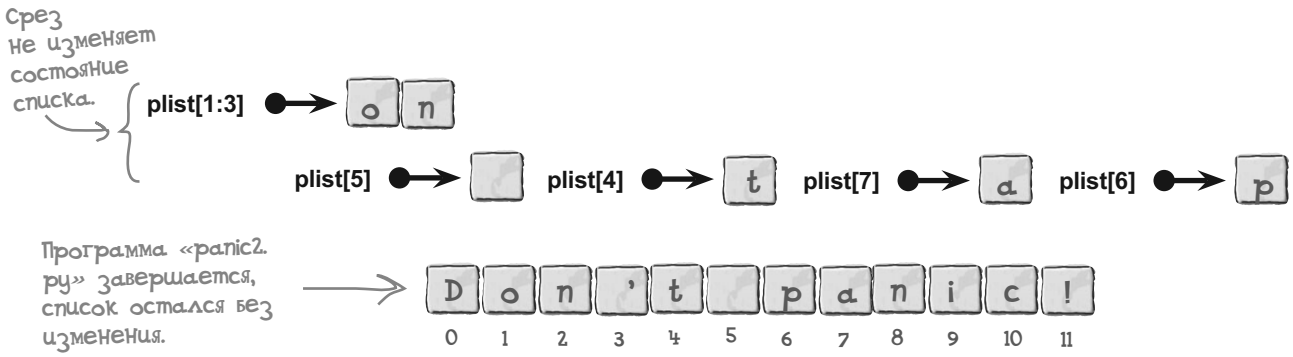
Методы списков, используемые в программе `panic.py` для превращения одной строки в другую, являются **деструктивными**, потому что изменяют состояние списка. Срезы **не изменяют список** — исходные данные остаются нетронутыми.



Срезы, используемые в `panic2.py`, показаны здесь. Обратите внимание, что извлечение данных из списка не изменяет сам список. Вот две строки кода, которые выполняют всю работу, также показаны данные, которые извлекает каждый срез:

Код.

```
new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])
```



Итак... какая из двух программ лучше?

Использование методов списков для изменения и преобразования существующего списка приводит к изменению *и* преобразованию списка. Начальное состояние списка больше не доступно в программе. В зависимости от того, что вы пытаетесь сделать, это может быть важно или не важно. Использование квадратных скобок обычно не изменяет существующий список, если вы только не присваиваете новое значение по определенному индексу. Использование срезов также не вносит изменения в список: данные остаются в начальном состоянии.

Какой из этих двух подходов «лучше», зависит от того, что вы пытаетесь сделать. Обычно задачу можно решить более чем одним способом. Списки в Python достаточно гибки и поддерживают различные способы работы с данными, которые они содержат.

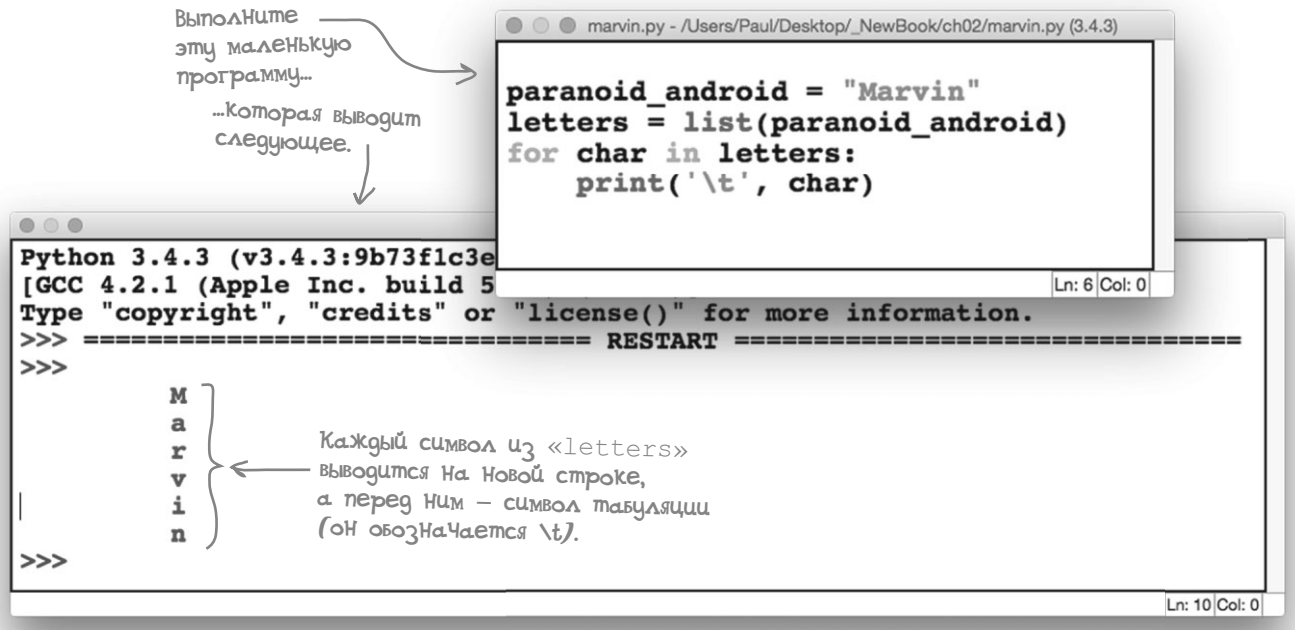
Мы уже сделали предварительный обзор списков. Осталось разобраться с *итерациями по спискам*.

Методы списков изменяют состояние данных, а квадратные скобки и срезы — нет (как правило).

Использование цикла «for» со списками в Python

Цикл `for` в Python знает все о списках, и когда получает *любой* список, он знает, где список начинается, сколько объектов в нем содержится и где список заканчивается. Эти сведения не нужно явно указывать циклу `for`, он определяет их сам.

Следующий пример это иллюстрирует. Откройте окно редактора и введите код. Затем сохраните программу под именем `marvin.py` и нажмите F5:

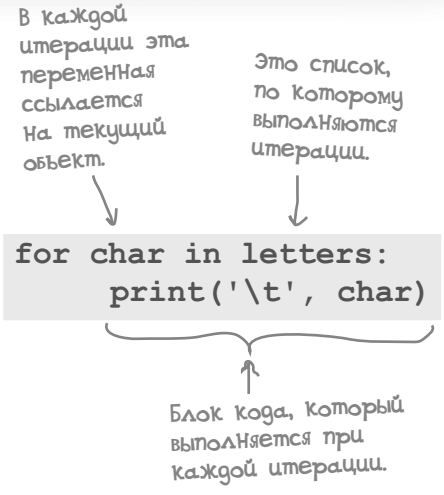


Как работает код marvin.py

Первые две строки в программе `marvin.py` вам знакомы: сначала переменной `paranoid_android` присваивается строка, а затем строка превращается в список символьных объектов, который присваивается новой переменной `letters`.

Следующую инструкцию — цикл `for` — рассмотрим подробнее.

В каждой итерации цикл `for` переходит к следующей букве в списке `letters` и присваивает ее переменной цикла `char`. В теле цикла переменная `char` принимает текущее значение объекта, который обрабатывается циклом `for`. Обратите внимание: цикл `for` знает, где *начать* итерации, где их *завершить* и *сколько* объектов в списке `letters`. Вам не нужно заботиться об этом, интерпретатор сам выполнит всю необходимую работу.



Цикл «for» умеет работать со срезами

При использовании нотации с квадратными скобками для выбора среза из списка цикл `for` будет выполнять итерации только по выбранным объектам. Как это сделать, мы покажем, модернизировав последний пример. Сохраните `marvin.py` в файле с именем `marvin2.py`, а затем измените код, как показано ниже.

Обратите внимание на **оператор умножения** `(*)`, который определяет, сколько символов табуляции поставить во втором и третьем циклах `for`.

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

Первый цикл организован по первым шести объектам в списке.

```
>>>
        M
        a
        r
        v
        i
        n

                A
                n
                d
                r
                o
                i
                d

                        P
                        a
                        r
                        a
                        n
                        o
                        i
                        d
>>>
```

Второй цикл выполняет обход среза из последних семи объектов списка. Обратите внимание, как «*2» вставляет два символа табуляции перед выводом каждого объекта.

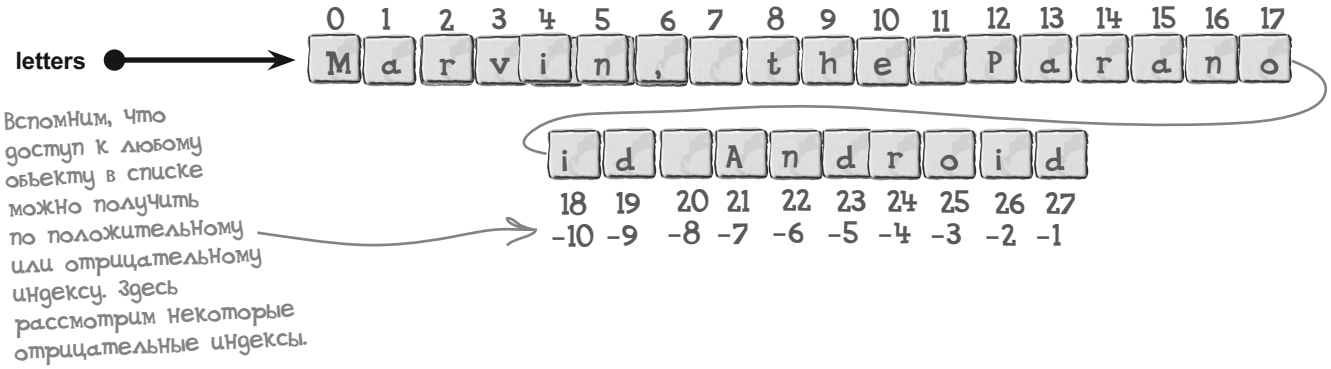
Третий цикл (и последний) выполняет обход среза со словом «Paranoid» из середины списка. Обратите внимание, что «*3» вставляет три символа табуляции перед выводом каждого объекта.

Срезы в деталях

Рассмотрим каждый срез, который использовался в предыдущей программе. Подобная техника часто используется в программах на Python. Ниже еще раз показаны строки кода, извлекающие срезы, дополненные графическим представлением.

Прежде чем посмотреть на все три среза, обратите внимание, что программа начинается с присваивания строки переменной (с именем `paranoid_android`) и преобразования строки в список (с именем `letters`):

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
```



Посмотрим на каждый из срезов в программе `marvin2.py` и разберемся, что они возвращают. Когда интерпретатор встречает спецификацию среза, он извлекает из списка `letters` объекты, указанные в описании среза, и возвращает их копию циклу `for`. При этом исходный список `letters` остается без изменений.

Первый срез начинается с первого элемента списка и заканчивается элементом (не включая) с индексом 6.

```
for char in letters[:6]:
    print('\t', char)
```

Второй срез начинается с индекса –7 и заканчивается в конце списка `letters`.

```
for char in letters[-7:]:
    print('\t'*2, char)
```

И наконец, третий срез извлекает объекты из середины списка, начиная с индекса 12 и заканчивая (но не включая) 20.

```
for char in letters[12:20]:
    print('\t'*3, char)
```

Списки: пополним список изученного

Теперь, когда нам известно, как взаимодействуют списки и циклы `for`, кратко перечислим все, что мы узнали на последних страницах.

КОНТРОЛЬНЫЙ СПИСОК

- Списки поддерживают нотацию с квадратными скобками для выбора отдельных объектов из списка.
- Как и в большинстве других языков, нумерация элементов в списках Python начинается с нуля, то есть первый элемент имеет индекс 0, второй 1 и т. д.
- В отличие от большинства других языков, Python позволяет индексировать списки с любого конца. Индекс –1 соответствует последнему элементу списка, –2 — предпоследнему и т. д.
- Списки также поддерживают срезы (или фрагменты), которые определяются как диапазоны (начало, конец, шаг) в квадратных скобках.

Списки используются часто, но...

Они *не* являются панацеей. Списки используются в большинстве случаев; если у вас есть коллекция похожих объектов, которые нужно хранить в структуре данных, списки — лучший выбор.

Возможно, это покажется нелогичным, но если данные, с которыми вы работаете, имеют свою внутреннюю *структуру*, списки могут оказаться **плохим выбором**. Мы объясним эту проблему (и возможные пути ее решения) на следующей странице.

Это не Глупые вопросы

В: Наверняка про списки можно узнать больше?

О: Да. Считайте эту главу введением во встроенные структуры данных Python и знакомством с их возможным применением. Мы не заканчиваем обсуждение списков и еще не раз будем возвращаться к ним в последующих главах.

В: А сортировка списков? Разве это не важно?

О: Да, но мы не будем рассматривать подобные вопросы, пока нам это не понадобится. Сейчас мы рассматриваем азы, на данном этапе этого достаточно. Не беспокойтесь: скоро мы рассмотрим и сортировку.

Какие проблемы могут возникнуть со списками?

Когда у программистов на Python возникает необходимость сохранить коллекцию похожих объектов, они обычно выбирают списки. Мы тоже до этого момента использовали только списки.

Вспомним еще раз, как удобны списки для хранения букв, например как в случае со списком `vowels`.

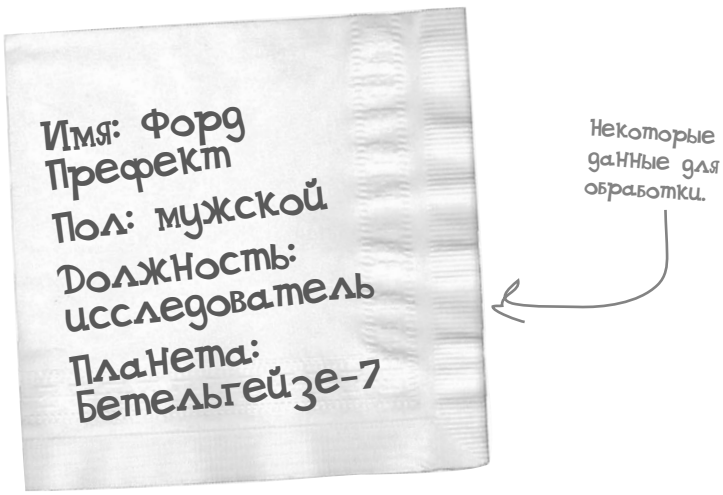
```
vowels = ['a', 'e', 'i', 'o', 'u']
```

Если нужно сохранить коллекцию числовых значений, то списки подойдут как нельзя лучше и в этом случае.

```
nums = [1, 2, 3, 4, 5]
```

Фактически списки — прекрасный выбор для хранения любых связанных объектов.

Однако представьте, что нужно сохранить данные о человеке, например такие:



На самом деле эти данные уже обладают некоторой структурой, потому что слева присутствует некоторый *тег*, а справа — *ассоциированное с ним значение*. Почему бы тогда не поместить эти данные в список? В конце концов, все эти данные связаны с некоторым человеком, не так ли?

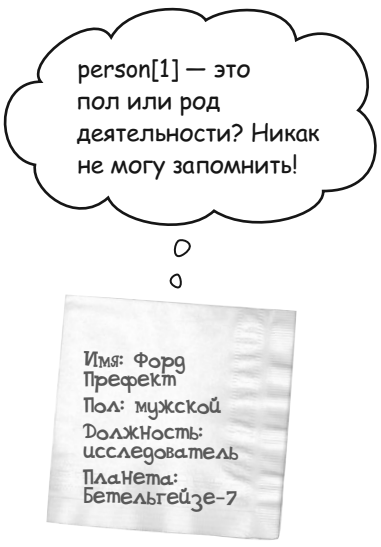
Чтобы понять, почему этого не нужно делать, рассмотрим два способа сохранить эти данные в списке (на следующей странице). Здесь мы потерпим поражение: *обе* попытки приведут к проблемам и покажут, что списки не так уж хороши для хранения таких данных, в отличие от связанных значений. Подумайте, как можно сохранить структурированные данные в списке, а затем переверните страницу и посмотрите на две неудачные попытки сделать это.

Когда не нужно использовать списки

У нас есть пример данных, и мы решили сохранить этот пример в списке (потому что на данном этапе изучения Python мы знаем только это).

В первой попытке просто возьмем значения и поместим их в список:

```
>>> person1 = ['Ford Prefect', 'Male',
'Researcher', 'Betelgeuse Seven']
>>> person1
['Ford Prefect', 'Male', 'Researcher',
'Betelgeuse Seven']
```



В результате получается список строковых объектов, и все работает. Как показано выше, вывод в консоли подтверждает, что данные сохранены в списке `person1`.

Однако проблема заключается в необходимости помнить, что под первым индексом (индекс `0`) хранится имя, затем следует пол (индекс `1`) и т. д. Когда элементов данных немного, это нестрашно, но представьте, что требуется гораздо большее количество элементов (например, для поддержки профиля в Facebook). С подобными данными списки использовать неудобно.

Попытаемся исправить ситуацию, добавив теги в список, чтобы каждый элемент данных предварялся соответствующим тегом. И у нас получился вот такой список `person2`.

```
>>> person2 = ['Name', 'Ford Prefect', 'Gender',
'Male', 'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
>>> person2
['Name', 'Ford Prefect', 'Gender', 'Male',
'Occupation', 'Researcher', 'Home Planet',
'Betelgeuse Seven']
```

Все это тоже работает, но теперь у нас не одна проблема, а сразу две. Нужно запоминать не только положение каждого индекса, нужно еще запомнить, что индексы 0, 2, 4, 6 и т. д. являются тегами, а индексы 1, 3, 5, 7 и т. д. являются значениями.

Но должен же быть лучший способ обработки таких структурированных данных?

Он есть. Это структура данных, которую можно использовать для хранения таких данных, как в нашем примере. Здесь нужно что-то другое, и в Python это «что-то» называется **словарем**. В главе 3 мы изучим **словари**.

Если данные, которые нужно хранить, имеют определенную структуру, подумайте об использовании другого контейнера вместо списков.

Код из 2-й главы, 1 из 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
for letter in word:
    if letter in vowels:
        print(letter)
```

Первая версия программы «vowels.py», которая выводит все гласные, найденные в слове «Milliways» (включая повторения).

Программа «vowels2.py», в которой не выводятся повторения. Программа выводит список неповторяющихся гласных, найденных в слове «Milliways».

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = "Milliways"
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Третья (и последняя) версия, программа «vowels3.py», которая выводит неповторяющиеся гласные, обнаруженные в слове, введенном пользователем.

А вот лучший совет для всего человечества «Don't panic!» (Не паникуй!). Программа «panic.py» принимает эту строку на входе и с помощью методов списков преобразует ее в строку с описанием любимого пива редакторов книги: «on tap» (разливное).

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

for i in range(4):
    plist.pop()
plist.pop(0)
plist.remove("'")
plist.extend([plist.pop(), plist.pop()])
plist.insert(2, plist.pop(3))

new_phrase = ''.join(plist)
print(plist)
print(new_phrase)
```

Код из 2-й главы, 2 из 2

```
phrase = "Don't panic!"
plist = list(phrase)
print(phrase)
print(plist)

new_phrase = ''.join(plist[1:3])
new_phrase = new_phrase + ''.join([plist[5], plist[4], plist[7], plist[6]])

print(plist)
print(new_phrase)
```

Методы списков — не единственный способ работы с ними. В программе «panic2.py» такой же результат получается за счет использования квадратных скобок.

```
paranoid_android = "Marvin"
letters = list(paranoid_android)
for char in letters:
    print('\t', char)
```

Самая короткая программа в разделе, «marvin.py», показывает, как хорошо списки в Python взаимодействуют с циклом «for». (Только Марвину не говорите. Если он узнает, что его программа — самая короткая, он станет еще большим параноиком, чем раньше.)

Программа «marvin2.py» демонстрирует использование квадратных скобок в Python на примере создания трех срезов для извлечения и отображения фрагментов из списка букв.

```
paranoid_android = "Marvin, the Paranoid Android"
letters = list(paranoid_android)
for char in letters[:6]:
    print('\t', char)
print()
for char in letters[-7:]:
    print('\t'*2, char)
print()
for char in letters[12:20]:
    print('\t'*3, char)
```

Работа со структурированными данными

Списки в Python очень удобны, но они не панацея.

Когда имеются *действительно* структурированные данные (для хранения которых список оказывается не лучшим выбором), спасение приходит от встроенных **словарей** Python. Словари «из коробки» позволяют хранить и обрабатывать данные, которые можно представить в виде *пар ключ/значение*. В этой главе мы изучим словари, а также **множества** и **кортежи**. Как и **списки** (которые мы изучили в главе 2), словари, множества и кортежи предоставляют встроенные инструменты для работы с данными, что делает Python еще более удобным языком.

Словари хранят пары ключ/значение

Словарь

В отличие от списков — коллекций связанных объектов, **словари** используются для хранения **пар ключ/значение**, где каждому уникальному *ключу* ставится в соответствие некоторое *значение*. Специалисты часто называют словари *ассоциативными массивами*; в других языках программирования используются другие названия словарей (например, отображение, хеш или таблица).

Обычно роль ключа в словаре Python играет строка, а ассоциированным значением может быть любой объект.

Данные, соответствующие словарной модели, легко узнать: это **много записей** с **двумя колонками** в каждой. Взяв это на заметку, взгляните еще раз на «салфетку с данными» из главы 2.

В C++ и Java словарь известен как «отображение» (map), а в Perl и Ruby используется название хеш (hash).

Похоже, что данные с салфетки прекрасно уложатся в словарь Python.

Используем консоль >>>, чтобы создать словарь для хранения этих данных. Есть соблазн создать словарь одной строкой кода, но мы поступим иначе. Мы хотим, чтоб наш код легко читался, поэтому специально будем вводить в словарь каждую запись с данными (то есть пару ключ/значение) отдельной строкой.

Имя словаря (напомним, что мы уже использовали имена «person1» и «person2» в конце главы 2).

Ассоциированное значение данных.

```
>>> person3 = { 'Name': 'Ford Prefect',
             'Gender': 'Male',
             'Occupation': 'Researcher',
             'Home Planet': 'Betelgeuse Seven' }
```

Ключ.

Значение.

Определяйте словари в легкочитаемом виде

Словарь

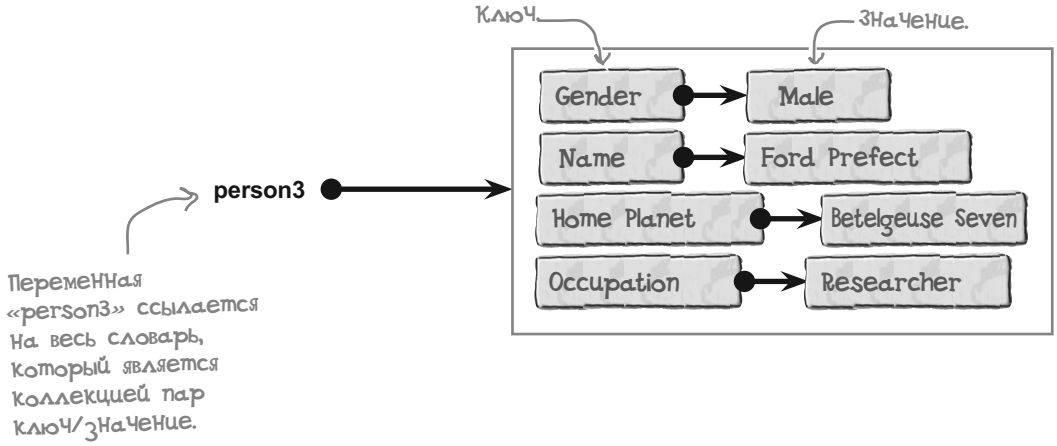
Иногда возникает соблазн записать все четыре строки кода, представленные выше, в одну строку.

```
>>> person3 = { 'Name': 'Ford Prefect', 'Gender':
'Male', 'Occupation': 'Researcher', 'Home Planet':
'Betelgeuse Seven' }
```

Интерпретатору безразлично, какой подход вы выберете, но определение словаря, записанное одной длинной строкой, читать труднее, поэтому от такого стиля лучше отказаться.

Если вы наводните свой код словарями, которые трудно читать, другие программисты (которые присоединятся к *вам* спустя шесть месяцев) будут огорчены... ведь им придется потратить время, чтобы упорядочить ваш код и сделать *его* легкочитаемым.

Вот как словари хранятся в памяти после выполнения любого из этих двух операторов присваивания.



Это более сложная структура данных, чем списки, похожие на массивы. Если раньше вы не сталкивались с подобными структурами, представляйте их как **таблицы поиска**. Ключ используется *для поиска* значения (как в обычных бумажных словарях).

Потратим немного времени, чтобы подробнее ознакомиться со словарями. Начнем с особенностей определения словарей в коде, а затем обсудим уникальные характеристики словарей и способы их использования.

Как определяются словари в коде

Словарь

Рассмотрим подробнее, как мы определили словарь `person3` в консоли >>>. Во-первых, *весь* словарь заключен в фигурные скобки. Все **ключи** взяты в кавычки, потому что являются строками, так же как и все **ассоциированные значения**, которые в этом примере тоже являются строками. (Однако ключи и значения необязательно должны быть строками.) Каждый ключ отделен от ассоциированного значения символом **двоеточия** (`:`), а каждая пара ключ/значение (то есть «запись») отделена от следующей **запятой**.

Определение любого словаря начинается с открывающей фигурной скобки.

В этом словаре значения являются строковыми объектами, поэтому они заключены в кавычки.

Пары ключ/значение отделяются друг от друга запятыми.

```
{ 'Name': 'Ford Prefect',
  'Gender': 'Male',
  'Occupation': 'Researcher',
  'Home Planet': 'Betelgeuse Seven' }
```

Каждый ключ помещен в кавычки.

Ключи и значения отделяются друг от друга двоеточиями.

Определение любого словаря завершается закрывающей фигурной скобкой.

Как говорилось ранее, словарь Python прекрасно подходит для хранения данных с салфетки. Фактически словари с успехом подойдут для хранения любых данных со схожей структурой — несколько записей с двумя колонками. Теперь поэкспериментируем в консоли.

Выведем содержимое словаря...

```
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home
Planet': 'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

...и вот оно. Все пары ключ/значение на экране.

Что произошло с порядком ввода?

Взгляните, как интерпретатор вывел содержимое словаря. Заметили, что порядок вывода отличается от порядка ввода? Когда мы создавали словарь, мы вводили имя (`'Name'`), пол (`'Gender'`), род занятий (`'Occupation'`) и планету (`'Home Planet'`). Порядок изменился.

Что произошло? Почему порядок изменился?

Порядок добавления НЕ поддерживается

В отличие от списков, которые хранят объекты в порядке их добавления, словари этого **не** делают. Это означает, что мы не можем делать какие-либо предположения относительно порядка следования записей в словарях; записи в словарях **не упорядочены** во всех смыслах.

Словарь

Еще раз взгляните на словарь `person3` и сравните порядок добавления с тем, что вернул интерпретатор.

Мы ввели данные в словарь в этом порядке...

...а интерпретатор использует другой порядок.

```
>>> person3 = { 'Name': 'Ford Prefect',
                'Gender': 'Male',
                'Occupation': 'Researcher',
                'Home Planet': 'Betelgeuse Seven' }
>>> person3
{'Gender': 'Male', 'Name': 'Ford Prefect', 'Home Planet':
'Betelgeuse Seven', 'Occupation': 'Researcher'}
```

Если вы ломаете голову и не понимаете, зачем доверять свои данные такой неупорядоченной структуре данных, не беспокойтесь, потому что порядок в данном случае не имеет значения. Если вы выбираете данные из словаря, то обычно делаете это по ключу. Запомните: ключ нужен для выбора значения.

Со словарями можно использовать квадратные скобки

Как и со списками, со словарями можно использовать квадратные скобки. Однако в отличие от списков, в которых для доступа к данным используются индексы, в словарях применяются ключевые значения. Посмотрим, как это можно сделать в консоли.

Используйте ключи для доступа к данным в словаре.

Укажите ключ между квадратными скобками.

```
>>> person3['Home Planet']
'Betelgeuse Seven'
>>> person3['Name']
'Ford Prefect'
```

Значение данных, ассоциированное с ключом.

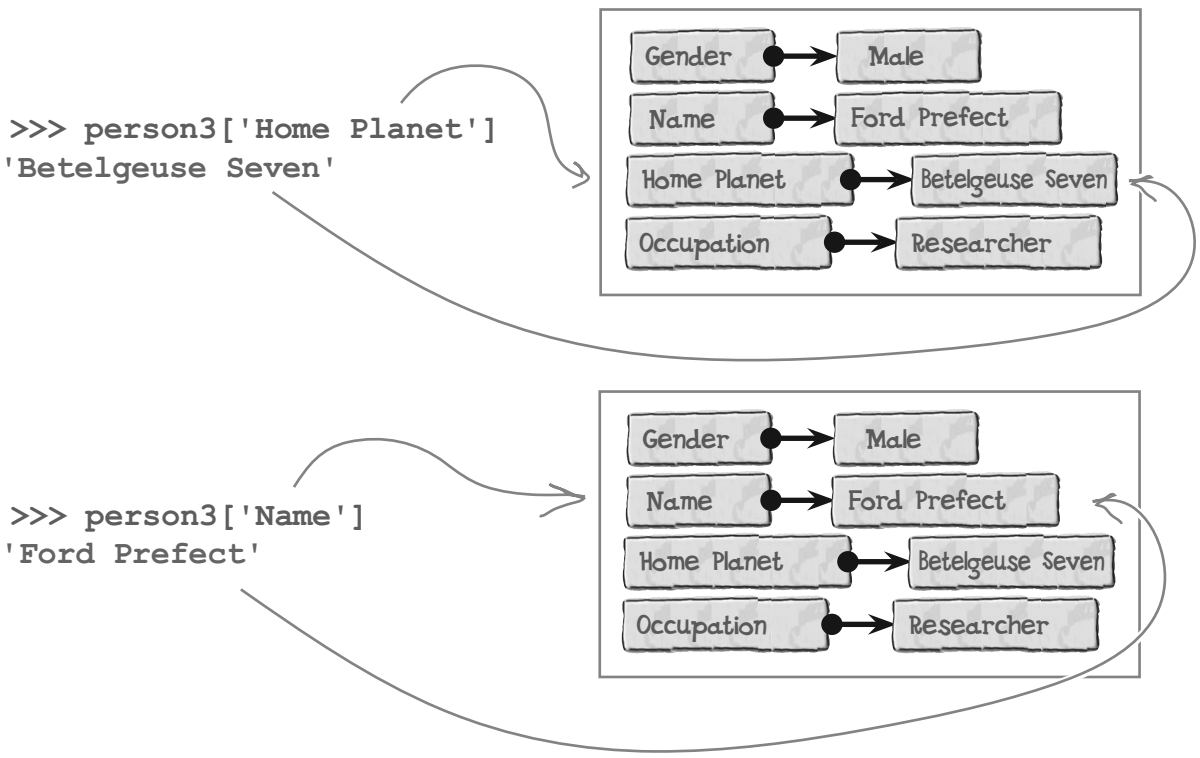
Когда вы поймете, что к данным можно обращаться подобным образом, станет ясно, что порядок не важен.

Выбор значений с помощью квадратных скобок

Словарь

Квадратные скобки можно использовать со словарями, как и со списками. Однако, в отличие от списков, доступ к данным в словарях производится по ключу, а не по индексу.

Как показано в конце прошлой страницы, интерпретатор возвращает значение данных, ассоциированное с ключом в квадратных скобках. Рассмотрим несколько примеров, чтобы закрепить эту идею.



Поиск данных в словарях работает быстро!

Возможность извлекать данные по ключу — очень полезная особенность словарей в Python. Она имеет массу практических применений, например извлечение информации из профиля пользователя. Именно это мы делали в словаре `person3`.

Не важно, в каком порядке хранятся данные в словаре. Все, что нам нужно, — это чтобы интерпретатор возвращал значение по ключу *быстро* (и не важно, как велик словарь). Самое замечательное, что интерпретатор прекрасно справляется с этой задачей, благодаря до предела оптимизированному *алгоритму хеширования*. Как в большинстве случаев при работе со встроенными структурами языка, интерпретатор выполнит за нас всю работу. А нам останется только наслаждаться преимуществами, которые предоставляет Python.

Для умников

Словари в Python реализованы как хеш-таблицы, оптимизированные для работы с особыми случаями. В результате поиск по словарям выполняется очень быстро.

Работа со словарями во время выполнения программы

Словарь

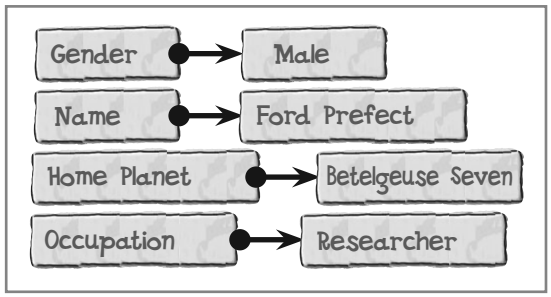
Знание того, как квадратные скобки работают со словарями, дает ключ к пониманию, как можно заполнять словари во время выполнения программы. Если у вас есть словарь, вы всегда сможете добавить в него новую пару ключ/значение, присвоив объект новому ключу, заключенному в квадратные скобки.

Например, выведем текущее состояние словаря `person3` и добавим новую пару ключ/значение, связав число `33` с ключом `'Age'` (Возраст). Затем снова выведем содержимое словаря `person3`, чтобы убедиться, что данные добавлены верно.

До добавления новой записи.

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

До...



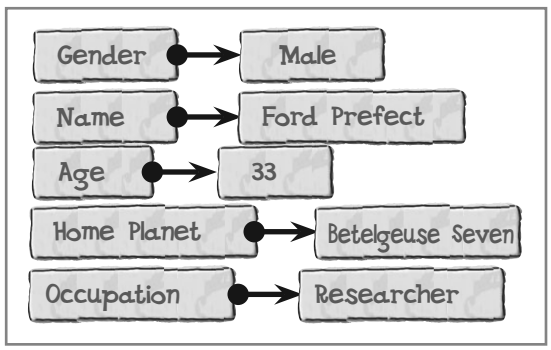

```
>>> person3['Age'] = 33
```

Присвоим объект (в данном случае число) новому ключу, чтобы добавить запись в словарь.

```
>>> person3
{'Name': 'Ford Prefect', 'Gender': 'Male',
'Age': 33, 'Home Planet': 'Betelgeuse Seven',
'Occupation': 'Researcher'}
```

После добавления записи.

Новая запись: значение «33» ассоциировано с ключом «Age».



После.

Вспомним: вывод найденных гласных (списки)

Возможность добавлять новые записи в словари, как показано на предыдущей странице, можно использовать в различных ситуациях. Одно из наиболее распространенных применений — *подсчет частоты*: обработка некоторых данных и подсчет найденного. Прежде чем продемонстрировать подсчет частоты с использованием словаря, вернемся к примеру с подсчетом гласных из прошлой главы.

Вспомним, что `vowels3.py` определяет список уникальных гласных, найденных в слове. Представьте, что нужно доработать программу, чтобы она сообщала, сколько раз каждая гласная обнаружена в слове.

Вот код из главы 2, который по заданному слову выводит список уникальных гласных.

Это программа «vowels3.py», которая выводит список гласных, найденных в слове, без повторов.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

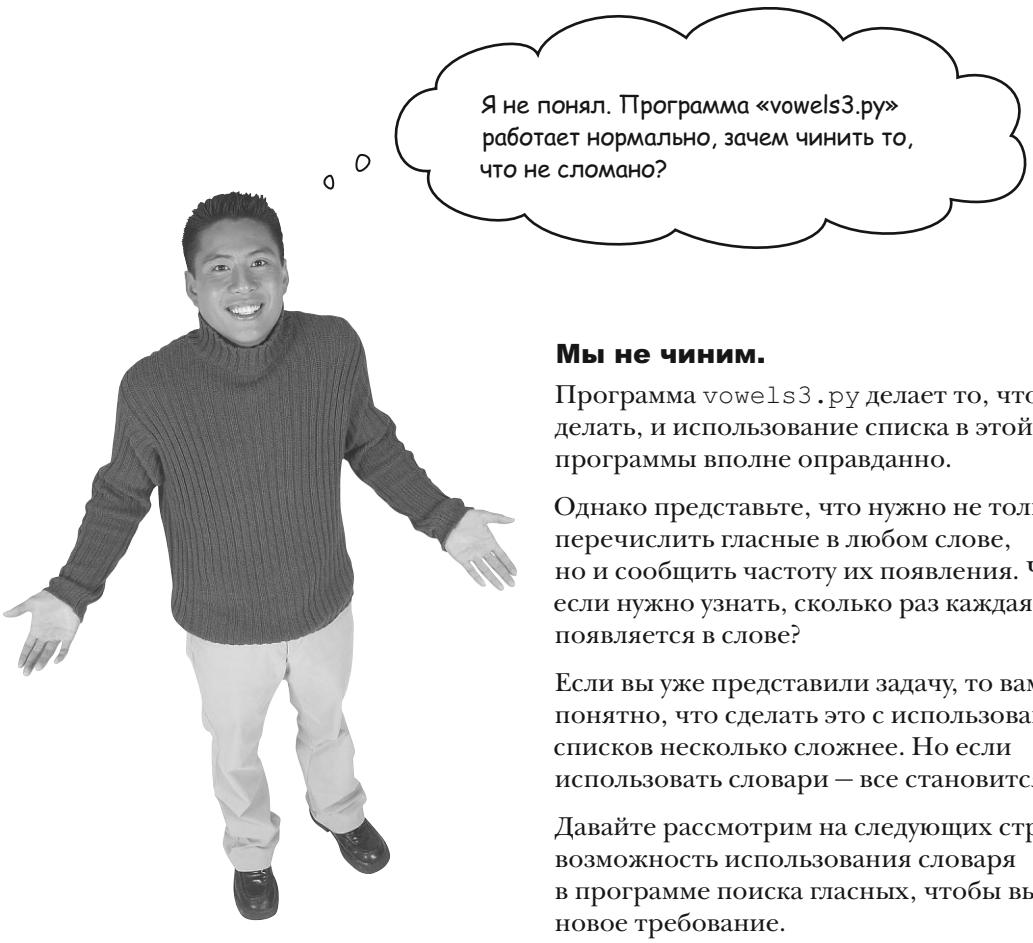
Вспомните, как мы запускали эту программу в IDLE несколько раз.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Milliways
i
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Hitch-hiker
i
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Sky
>>>
```

Ln: 21 Col: 4

Как здесь может помочь словарь?



Мы не чиним.

Программа `vowels3.py` делает то, что должна делать, и использование списка в этой версии программы вполне оправданно.

Однако представьте, что нужно не только перечислить гласные в любом слове, но и сообщить частоту их появления. Что, если нужно узнать, сколько раз каждая гласная появляется в слове?

Если вы уже представили задачу, то вам понятно, что сделать это с использованием списков несколько сложнее. Но если использовать словари — все становится проще.

Давайте рассмотрим на следующих страницах возможность использования словаря в программе поиска гласных, чтобы выполнить новое требование.

Это не Глупые вопросы

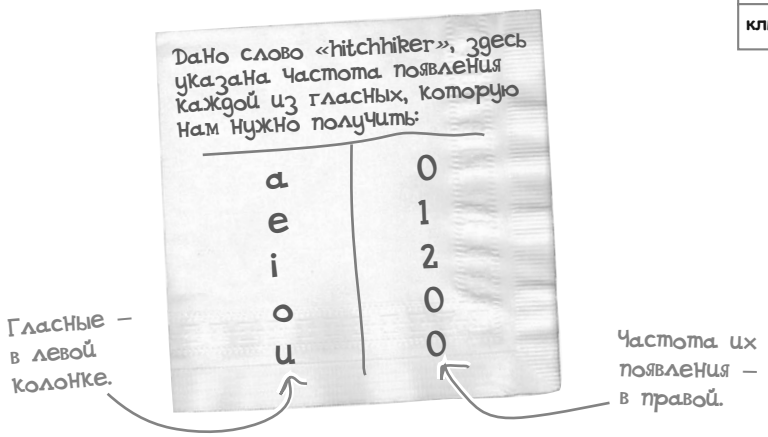
В: **Это только я считаю, что слово «словарь» — странное обозначение того, что изначально является таблицей?**

О: Не только вы. Слово «словарь» (dictionary) используется в документации Python. Фактически большинство программистов Python используют более короткое название «dict». В простейшей форме словарь — это таблица с двумя колонками и произвольным количеством строк.

Выбор структуры данных для подсчета частоты

Словарь

Преобразуем программу `vowels3.py` так, чтобы она подсчитывала количество появлений каждой гласной в заданном слове, определяя их частоту. Проанализируем, что мы собираемся получить на выходе.



Этот набросок точно отражает, как интерпретатор воспринимает словари. Попробуем вместо списка (как в случае `vowels3.py`) использовать для хранения найденных гласных словарь. Коллекцию найденных гласных снова назовем `found`, но инициализируем не пустым списком, а пустым словарем.

Как обычно, поэкспериментируем в консоли и выясним, что нужно сделать, прежде чем вносить изменения в код `vowels3.py`. Чтобы создать пустой словарь, присвойте `{}` переменной.

```
>>> found = {}
>>> found
{}
```

Фигурные скобки обозначают пустой словарь.

Зафиксируем тот факт, что пока не найдена ни одна гласная, создав для каждой свою запись и ассоциировав ключи с нулевыми значениями. Роль ключей будут играть сами гласные.

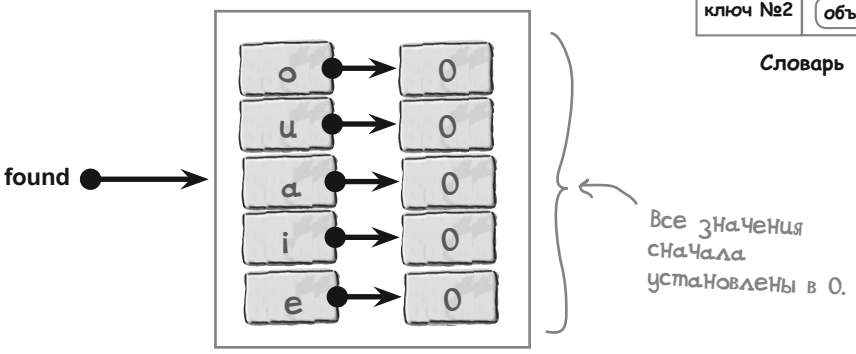
```
>>> found['a'] = 0
>>> found['e'] = 0
>>> found['i'] = 0
>>> found['o'] = 0
>>> found['u'] = 0
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
```

Мы инициализировали все счетчики гласных значением 0. Отметьте, что порядок создания записей не сохранился (но он и не важен).

Теперь нам нужно найти гласные в заданном слове и изменить соответствующие счетчики.

Изменение счетчика

Прежде чем углубиться в код, изменяющий счетчики, рассмотрим, как выглядит словарь `found` в памяти после его инициализации.



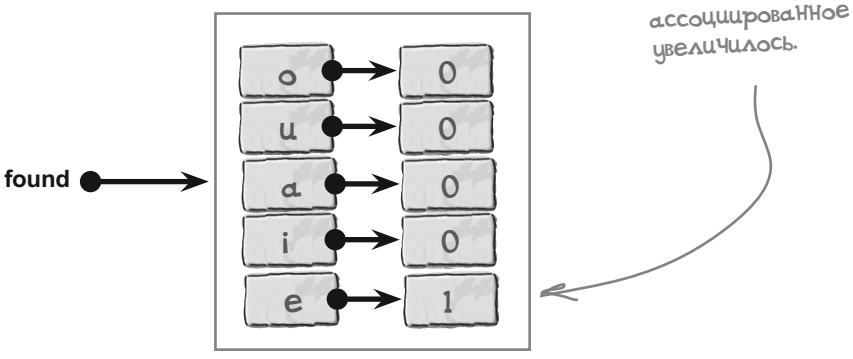
Когда все счетчики установлены в 0, не составляет труда увеличить любой из них при необходимости. Например, вот как можно увеличить счетчик частоты появления `'e'`.

```
>>> found
{'o': 0, 'u': 0, 'a': 0, 'i': 0, 'e': 0}
>>> found['e'] = found['e'] + 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 1}
```

Все значения равны 0.

Увеличивает счетчик, ассоциированный с ключом `'e'`.

Словарь изменился. Значение, ассоциированное с «e», увеличилось.



Код, выделенный на картинке, работает, но приходится повторять `found['e']` с обеих сторон от оператора присваивания. Посмотрим, можно ли записать эту операцию короче (на следующей странице).

Обновление счетчика частоты, версия 2.0

Словарь

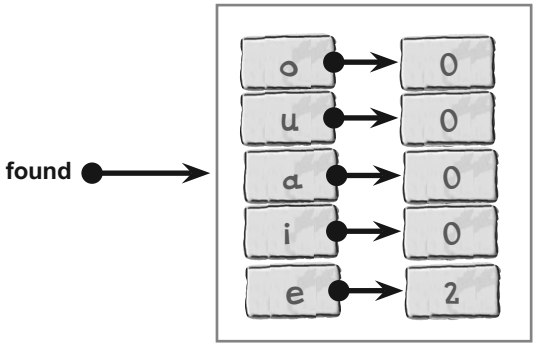
Необходимость повторять `found['e']` с обеих сторон от оператора присваивания (`=`) быстро утомляет, однако Python поддерживает схожий оператор `+=`, который выполняет то же самое, но запись получается более короткой.

```
>>> found['e'] += 1
>>> found
{'o': 0, 'i': 0, 'a': 0, 'u': 0, 'e': 2}
```

Увеличение счетчика 'e' (еще раз).

Словарь снова изменился.

К данному моменту значение, ассоциированное с ключом `'e'`, было увеличено уже дважды, поэтому теперь словарь выглядит так.



Благодаря оператору += значение, ассоциированное с ключом 'e', увеличилось еще раз.

Глупые вопросы (это не)

В: Имеется ли в Python оператор `++`?

О: Нет… такая вот неприятность. Если вы поклонник оператора инкремента `++`, имеющегося в других языках программирования, вам придется использовать `+=` вместо него. Так же обстоит дело с оператором декремента (`--`): в Python его нет. Используйте `-=`.

В: Есть ли удобный список операторов?

О: Да. Посмотрите на *https://docs.python.org/3/reference/lexical_analysis.html#operators*. Подробное описание их использования со встроенными типами дано на *https://docs.python.org/3/library/stdtypes.html*.

Итерации по записям в словарях

Словарь

Теперь вы знаете, как инициализировать словарь нулевыми значениями и как изменять значения в словаре по ключу. Мы почти готовы изменить программу `vowels3.py`, чтобы она считала частоту появления каждой из гласных в слове. Однако прежде посмотрим, как выполнять итерации по словарю. Поскольку словари оперируют данными, нам нужен способ вывода данных на экран.

Вы могли подумать, что достаточно воспользоваться циклом `for`, однако в этом случае возможны непредсказуемые результаты.

Выполняем итерации по записям в словаре обычным способом, используя цикл «for». Имя «kv» выбрано как сокращение от «key/value» (ключ/значение), но вообще можно использовать любое имя переменной.

```
>>> for kv in found:
        print(kv)

o
i
a
u
e
```

Итерации выполняются, но выводится не то, что мы ожидали. Куда делись подсчитанные частоты? На выходе только ключи...

Переверните страницу, чтобы узнать, что произошло со значениями.

Итерации по ключам и значениям

Словарь

Выполняя итерации по словарю в цикле `for`, интерпретатор обрабатывает только ключи.

Чтобы получить доступ к ассоциированным значениям, нужно поместить каждый ключ в квадратные скобки и использовать его вместе с именем словаря.

Версия цикла, который это выполняет, показана ниже. Теперь обрабатываются не только ключи, но и связанные с ними значения. Мы немного изменили код, чтобы получить доступ к данным и ключам внутри цикла `for`.

Выполняя итерации по парам ключ/значение, цикл `for` присваивает ключ текущей записи переменной `k`, а затем для доступа к значению используется `found[k]`. Кроме того, мы выводим данные в более дружественной форме, передавая обе строки в вызов функции `print`.

Используем переменную «k» для представления ключа, а «found[k]» ссылается на ассоциированное с ключом значение.

```
>>> for k in found:
        print(k, 'was found', found[k], 'time(s).')

o was found 0 time(s).
i was found 0 time(s).
a was found 0 time(s).
u was found 0 time(s).
e was found 2 time(s).
```

Теперь ключи и значения обрабатываются циклом и отображаются на экране.

Если вы выполняли все примеры в консоли >>>, а вывод отличается от нашего, не беспокойтесь: интерпретатор хранит записи в словарях не в порядке добавления и не дает никаких гарантий относительно порядка их следования. Если порядок вывода у вас отличается от нашего, в этом нет ничего плохого. Самое главное, что данные правильно сохранены в словаре, и на этом все.

Предыдущий цикл, *безусловно*, работает. Однако есть еще две детали, которые нужно проработать.

Во-первых, было бы неплохо, если бы порядок вывода был именно `a`, `e`, `i`, `o`, `u`, а не случайный.

Во-вторых, хотя этот цикл работает, такой способ итераций по словарям — не самый удобный, большинство программистов Python пишут код иначе.

После краткого обзора рассмотрим оба замечания подробнее.

Словари: что мы уже знаем

Вот что мы узнали о словарях в Python.

КОНТРОЛЬНЫЙ СПИСОК

- Словари можно считать коллекцией записей, каждая из которых содержит два столбца. Первый столбец — **ключ**, второй — **значение**.
- Каждая запись — это **пара ключ/значение**, и в словарь можно добавить произвольное количество таких пар. Подобно спискам, словари позволяют добавлять или удалять значения по необходимости.
- Словарь в коде легко заметить: он заключен в фигурные скобки, каждая пара ключ/значение отделена от последующей запятой, а каждый ключ отделен от значения двоеточием.
- Порядок вставки *не* поддерживается словарями. Порядок добавления записей в словарь не имеет ничего общего с порядком их хранения.
- Для доступа к данным в словаре используются **квадратные скобки**, между которыми помещается ключ.
- Цикл `for` может работать со словарем. В каждой итерации ключ присваивается переменной цикла, которую можно использовать для доступа к данным.

Меняем порядок вывода записей из словаря

Мы хотели бы вывести гласные в цикле `for` в порядке `a`, `e`, `i`, `o`, `u`, а не случайным образом. В Python сделать это очень просто, благодаря встроенной функции `sorted`. Просто передайте словарь в функцию `sorted` внутри цикла `for`, чтобы получить вывод в алфавитном порядке.

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

Небольшое изменение в цикле и... все получилось. Взгляните: вывод отсортирован в алфавитном порядке.

Это первый из двух пунктов. Теперь научимся подходу, *предпочтительному* для большинства программистов Python (подход, описанный на этой странице, также часто используется, и его нужно знать).

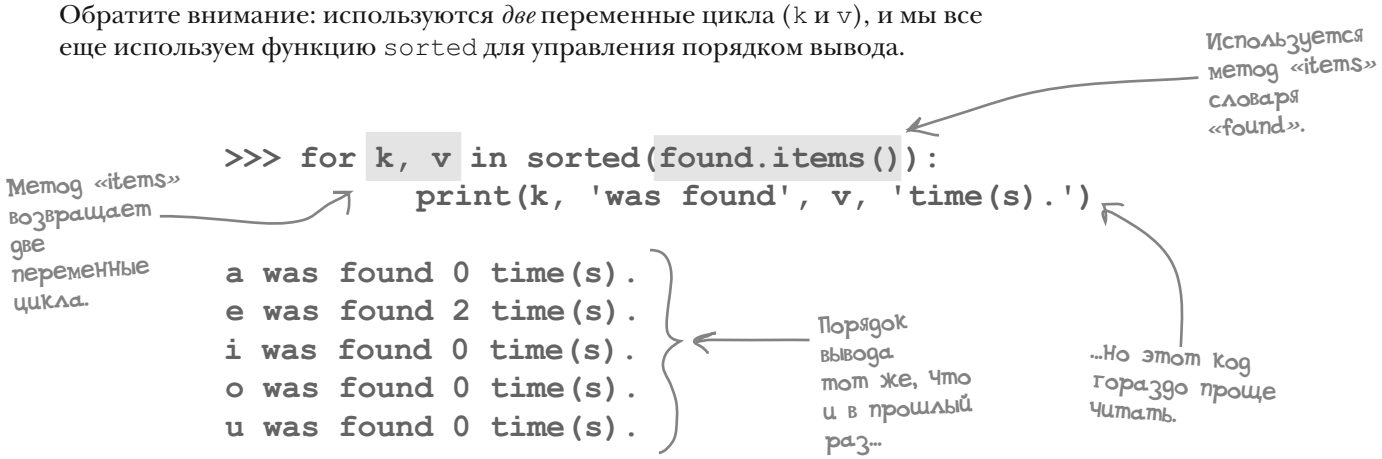
Итерации по словарям с использованием items

Мы видели, что итерации по записям в словаре можно реализовать так.

```
>>> for k in sorted(found):
        print(k, 'was found', found[k], 'time(s).')

a was found 0 time(s).
e was found 2 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
```

Как и списки, словари обладают набором встроенных методов, и один из этих методов — `items` — возвращает список пар ключ/значение. Использование `items` в цикле `for` — наиболее предпочтительный способ итераций по словарям, потому что в данном случае *и* ключ, *и* значение являются переменными цикла, которые сразу готовы к использованию. Результирующий код проще выглядит и легче читается.

Вот эквивалент предыдущего кода, в котором используется `items`. Обратите внимание: используются *две* переменные цикла (`k` и `v`), и мы все еще используем функцию `sorted` для управления порядком вывода.



Это не Глупые вопросы

В: Почему во втором цикле снова использована функция `sorted`? В первом цикле мы уже отсортировали словарь в нужном порядке, разве это не означает, что его больше не нужно сортировать?

О: Не совсем. Функция `sorted` не меняет порядок следования записей в самой структуре данных, а просто возвращает **отсортированную копию**. В этом случае — копию словаря `found`, в которой пары ключ/значение отсортированы по ключам в алфавитном порядке (от A до Z). Порядок в исходном словаре не меняется, поэтому мы должны выполнять сортировку каждый раз, когда хотим обрабатывать пары ключ/значение в каком-то определенном порядке.

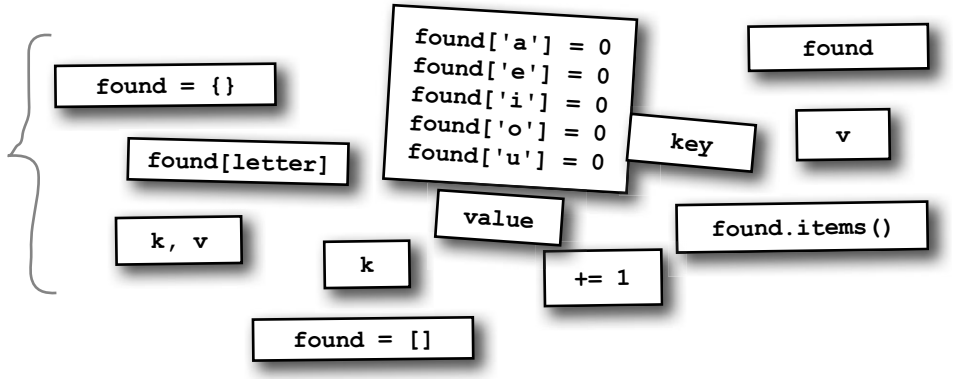
Магнитики для подсчета частоты

После экспериментов в консоли пришло время внести изменения в программу `vowels3.py`. Внизу представлены отрывки кода, которые могут вам пригодиться. Ваша задача — расставить фрагменты кода так, чтобы получить действующую программу, подсчитывающую частоту использования каждой из гласных во введенном слове.

Определите, какой магнитик должен быть в каждой из пустых строк, чтобы получилась программа «vowels4.py».

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

.....
.....
.....
.....
.....
.....
.....
for letter in word:
    if letter in vowels:
        ..... .....

for ..... in sorted( ..... ):
    print( ..... , 'was found', ..... , 'time(s).')
```

Куда их все поместить? Будьте внимательны: не все эти магнитики нужны.



Поместив магнитики туда, где, по вашему мнению, они должны быть, перейдите в окно редактирования IDLE, сохраните программу `vowels3.py` с именем `vowels4.py` и внесите необходимые изменения.

Решение задачи подсчета частоты

После экспериментов в консоли пришло время внести изменения в программу `vowels3.py`. Вы должны были расположить магнитики так, чтобы получить действующую программу, подсчитывающую частоту использования каждой из гласных во введенном слове.

Разместив магнитики по своим местам, вы должны были переименовать программу `vowels3.py` в `vowels4.py` и внести необходимые изменения в новую версию программы.

Это программа «vowels4.py».

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
```

Создать пустой словарь.

```
found = {}
```

Инициализировать нулями значения, ассоциированные с каждым из ключей.

```
found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0
```

```
for letter in word:
    if letter in vowels:
        found[letter] += 1
```

Увеличить значение «found[letter]» на единицу.

```
for k, v in sorted( found.items() ):
    print( k , 'was found', v , 'time(s).')
```

Так как в цикле «for» используется метод «items», мы должны определить две переменные цикла: «k» — для ключей, «v» — для значений.

Ключ и значение используются для вывода сообщений.

Вызвать метод «items» словаря «found», чтобы получить доступ к каждой записи с данными во время итераций.

Эти магнитики не нужны.

```
found
key
value
found = []
```

Пробная поездка

Теперь запустим `vowels4.py`. Выбрав окно редактора IDLE, нажмите F5, чтобы увидеть результаты выполнения:

Код «vowels4.py».

vowels4.py - /Users/Paul/Desktop/_NewBook/ch02/vowels4.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Мы выполнили код три раза, чтобы убедиться в его работоспособности.

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
a was found 0 time(s).
e was found 1 time(s).
i was found 2 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
o was found 0 time(s).
u was found 1 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: sky
a was found 0 time(s).
e was found 0 time(s).
i was found 0 time(s).
o was found 0 time(s).
u was found 0 time(s).
>>>
```

Эти три «попытки» дали ожидаемые результаты.

Меня все устраивает. Но зачем мне знать, что какие-то гласные не были найдены?

Насколько динамичны словари?

Программа `vowels4.py` сообщает результаты поиска всех гласных, даже отсутствующих в слове. Возможно, в этом нет ничего страшного, но представьте, что вам нужно изменить код, чтобы выводились только результаты для *действительно* найденных значений. Допустим, вы не хотите видеть никаких сообщений типа «`found 0 time(s)`».

Как решить проблему?

Словари Python — динамические структуры, верно? Значит, нужно просто удалить пять строк, которые инициализируют счетчики частоты. Так мы подсчитаем только найденные гласные, да?

Похоже, это может сработать.

Сейчас в начале программы `vowels4.py` есть пять строк кода, которые устанавливают *начальные* нулевые значения счетчиков частоты появления гласных. Если мы уберем эти строки, программа будет записывать только частоты найденных гласных и игнорировать остальные.

Давайте попробуем.

Сделай это!

Сохраните код `vowels4.py` в файле с именем `vowels5.py`. Затем удалите пять строк кода инициализации. Ваше окно редактора IDLE должно выглядеть так:

Это программа «vowels5.py», из которой удален код инициализации.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Ln: 13 Col: 0

Пробная поездка

Вы уже знаете, что делать. Убедитесь, что программа `vowels5.py` открыта в окне редактора IDLE, а затем нажмите F5, чтобы запустить программу. Вы увидите сообщение об ошибке времени выполнения.

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
                                                                  Ln: 11 Col: 0
```

Это не здорово.

Очевидно, что удаление пяти строк инициализации — плохая идея. Но что вызвало ошибку? Словари Python могут расти динамически, поэтому этот код *не должен* завершаться аварийно, но ошибка налицо. Почему она возникла?

Ключи в словарях должны быть инициализированы

Удаление кода инициализации привело к ошибке времени выполнения `KeyError`, которая возникает при попытке получить доступ к значению по несуществующему ключу. Поскольку ключ отсутствует, ассоциированное с ним значение также отсутствует, и мы получаем ошибку.

Означает ли это, что мы должны вернуть код инициализации? В конце концов, это всего лишь пять безвредных строчек. Конечно, можно так поступить, но давайте сначала подумаем.

Представьте, что вместо пяти нам нужны тысячи счетчиков частоты (или больше). Ни с того ни с сего нам придется написать *очень много* кода инициализации. Мы можем «автоматизировать» инициализацию с помощью цикла, но по-прежнему должны создать огромный словарь с большим количеством записей, многие из которых нам не понадобятся.

Если бы только был способ создания пар ключ/значения «на лету», потому что он нам нужен.

Для умников

Альтернативное решение проблемы заключается в обработке исключения времени выполнения (в нашем примере «KeyError»). Мы обсудим способы обработки исключений в Python немного позже, а пока наберитесь терпения.

Хороший вопрос.

Впервые мы встретили `in`, когда проверяли наличие значения в списке. Возможно, `in` со словарями тоже работает?

Давайте поэкспериментируем в консоли.

Предотвращение ошибок KeyError во время выполнения

Словарь

Как и в случае со списками, оператор `in` позволяет проверить наличие ключа в словаре; в зависимости от результата поиска интерпретатор вернет `True` или `False`.

Попробуем использовать это, чтобы избежать исключения `KeyError`, потому что очень неприятно, когда код вдруг прекращает работу и появляется сообщение об ошибке.

Для демонстрации создадим словарь `fruits`, а затем используем оператор `in`, чтобы избежать ошибки `KeyError` при обращении к несуществующему ключу. Сначала создадим пустой словарь; затем добавим пару ключ/значение, ассоциирующую ключ `apples` (яблоки) со значением `10`. После этого используем оператор `in`, чтобы убедиться, что ключ `apples` действительно существует.

```
>>> fruits
{}
>>> fruits['apples'] = 10
>>> fruits
{'apples': 10}
>>> 'apples' in fruits
True
```

Все, как мы и ожидали. Значение связано с ключом, и при использовании оператора «in» для проверки существования ключа никакие ошибки не появились.

Прежде чем сделать что-то еще, посмотрим, как интерпретатор хранит в памяти словарь `fruits`.

Это не Глупые вопросы

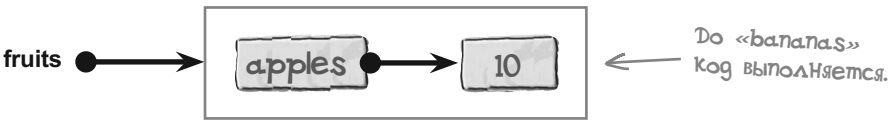
В: Я понял из примера, что Python использует константу `True` для обозначения истины? Существует ли константа `False` и есть ли смысл использовать эти значения?

О: Да — на все вопросы. Если в программе на Python потребуется указать булево значение, используйте `True` или `False`. Это предопределенные константы и должны начинаться с заглавной буквы, поскольку `true` и `false` интерпретатор будет рассматривать как имена переменных, а не булевы значения.

Проверка вхождения с помощью in

Словарь

Добавим еще одну запись в словарь `fruits` — с ключом `bananas` — и посмотрим, что из этого получится. Однако вместо прямого присваивания значения ключу `bananas` (как в случае с `apples`), попробуем увеличить ассоциированное значение на `1`, если ключ уже имеется в словаре `fruits`, иначе просто инициализируем `bananas` значением `1`. Такое поведение очень распространено, особенно когда словарь используется для подсчета частоты появления значений. Эта логика поможет нам избежать `KeyError`.



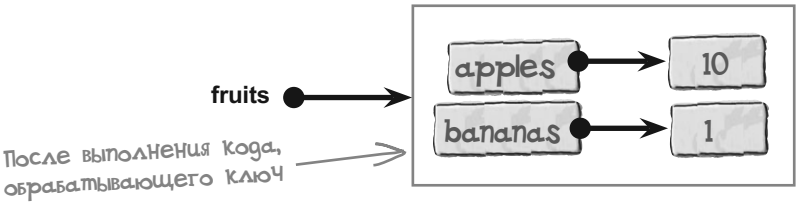
В следующем фрагменте кода оператор `in` совместно с инструкцией `if` предотвращает появление каких-либо ошибок при обращении к `bananas`.

```
>>> if 'bananas' in fruits:
        fruits['bananas'] += 1
else:
        fruits['bananas'] = 1

>>> fruits
{'bananas': 1, 'apples': 10}
```

Проверяем присутствие ключа «bananas» в словаре, и если он отсутствует, инициализируем значением 1. Так мы предотвращаем появление «KeyError».

Ключ «bananas» получил значение 1.

Код выше изменяет состояние словаря `fruits` в памяти интерпретатора, как показано здесь.



Как и предполагалось, словарь `fruits` вырос на одну пару ключ/значение, теперь ключ `bananas` инициализирован значением `1`. Это произошло потому, что условное выражение в инструкции `if` вернуло `False` (ключ не был найден) и выполнился второй блок кода (ветка `else`). Посмотрим, что получится, если выполнить код еще раз.

Для умников

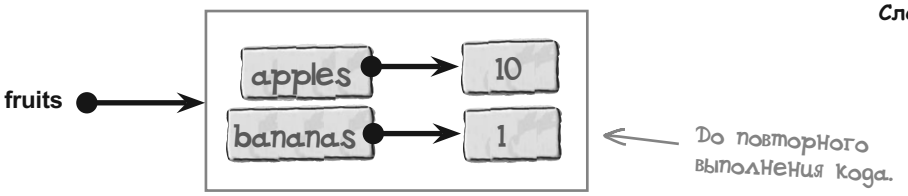
Если вы знакомы с **тернарным оператором** **? :** из других языков программирования, обратите внимание, что Python тоже поддерживает подобный оператор. Вы можете записать

```
x = 10 if y > 3 else 20
```

и присвоить переменной **x** значение **10**, если **y** больше **3**, и **20** — в противном случае. Однако большинство программистов на Python предпочитают использовать инструкцию **if... else...** которая проще читается.

Не забывайте инициализировать ключ перед использованием

Если мы выполним этот код снова, значение, ассоциированное с ключом `bananas`, увеличится на `1`, потому что выполнится ветка `if`, так как ключ `bananas` уже существует в словаре `fruits`.



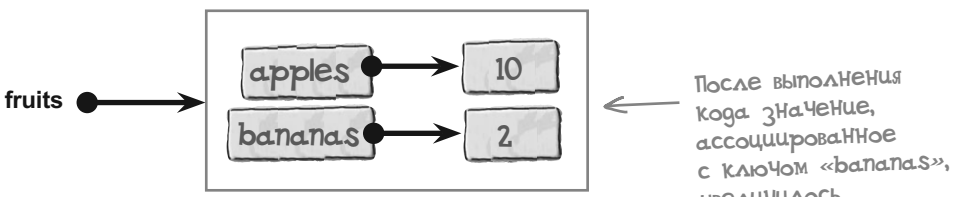
Чтобы выполнить код снова, нажмите *Ctrl-P* (в *Mac*) или *Alt-P* (в *Linux/Windows*), и вы вернетесь к прежде введенной инструкции (клавиши со стрелками в IDLE не работают). Не забудьте *дважды* нажать Enter, чтобы еще раз выполнить код.

```
>>> if 'bananas' in fruits:
      fruits['bananas'] += 1
else:
      fruits['bananas'] = 1

>>> fruits
{'bananas': 2, 'apples': 10}
```

Теперь ключ «bananas» есть в словаре, поэтому ассоциированное с ним значение увеличится на 1. Как и в предыдущем случае, «if» и «in» вместе предотвращают появление исключения «KeyError».

Мы увеличили на 1 значение, ассоциированное с «bananas».

Поскольку теперь выполняется код в ветке `if`, значение, связанное с `bananas`, увеличивается.



Этот прием получил настолько широкое распространение, что многие программисты на Python сокращают данный код, инвертируя условие. Вместо `in` они используют проверку `not in`. Это позволяет инициализировать код начальным значением (обычно `0`), если ключ не найден, а затем сразу выполнить инкремент.

Посмотрим, как работает этот механизм.

Замена in на not in

Словарь

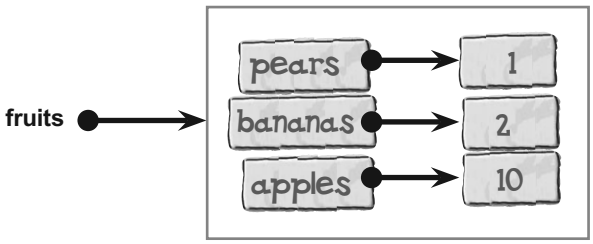
Как мы и говорили на прошлой странице, начнем с того, что отредактируем код, заменив `in` на `not in`. Посмотрим, как данная замена поможет проверить наличие ключа `pears` перед тем, как его увеличить.

```
>>> if 'pears' not in fruits:
        fruits['pears'] = 0

>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 1, 'apples': 10}
```

Инициализировать (если нужно).

Увеличить.

Эти три строки кода снова увеличили размер словаря. Теперь в словаре `fruits` три пары ключ/значение.



После выполнения трех строк, добавивших ключ «pears».

Эти три строки кода получили настолько широкое распространение, что в язык Python был даже включен метод словарей, реализующий комбинацию `if/not in` более удобным и более устойчивым к ошибкам способом. Метод `setdefault` выполняет то же, что и две строки кода с инструкциями `if/not in`, но теперь можно обойтись *одной* строкой.

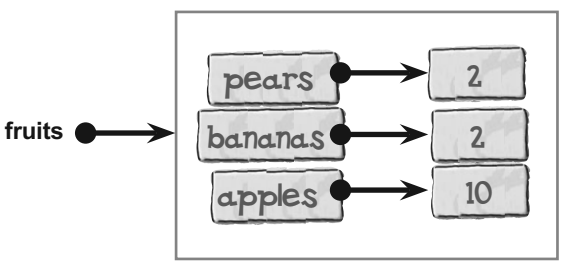
Вот эквивалент кода для `pears` с использованием метода `setdefault`.

```
>>> fruits.setdefault('pears', 0)
>>> fruits['pears'] += 1
>>> fruits
{'bananas': 2, 'pears': 2, 'apples': 10}
```

Инициализировать (если нужно).

Увеличить.

Единственный вызов `setdefault` позволил заменить двухстрочную инструкцию `if/not in`; он гарантирует, что ключ всегда будет инициализирован до использования. Устранена любая возможность появления исключения `KeyError`. Справа показано текущее состояние словаря `fruits`, чтобы подтвердить, что для существующего ключа `setdefault` не выполняет никаких действий (в данном случае с `pears`), как мы и хотели.



Работа с методом setdefault

Как вы помните, наша текущая версия `vowels5.py` вызывает ошибку `KeyError`, которая появляется из-за обращения по несуществующему ключу.

Словарь

Этот код вызывает ошибку.

vowels5.py - /Users/Paul/Desktop/_NewBook/ch03/vowels5.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitchhiker
Traceback (most recent call last):
  File "/Users/Paul/Desktop/_NewBook/ch03/vowels5.py", line 9, in <module>
    found[letter] += 1
KeyError: 'i'
>>>
```

Ln: 11 Col: 0

В ходе экспериментов с `fruits` мы узнали, что метод `setdefault` можно вызывать, сколько потребуется, не беспокоясь о неприятностях. Также мы узнали, что `setdefault` гарантирует инициализацию несуществующих ключей, а если ключ существует, то вызов метода не изменяет состояния словаря (то есть оставляет текущее значение, ассоциированное с ключом, без изменений). Вызвав метод `setdefault` перед использованием ключа в программе `vowels5.py`, мы гарантированно избежим появления `KeyError`. Теперь наша программа будет работать и перестанет завершаться аварийно (благодаря использованию метода `setdefault`).

Использование «setdefalut» помогает избежать появления «KeyError».

В окне редактора IDLE внесите изменения в код `vowels5.py` (добавив вызов `setdefault`), чтобы цикл `for` выглядел как показано ниже, и сохраните программу под именем `vowels6.py`.

```
for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1
```

Единственная строка кода очень часто может внести большие изменения в программу.

Пробная поездка

Открыв последнюю версию программы `vowels6.py` в окне редактора IDLE, нажмите F5. Запустите эту версию несколько раз, чтобы убедиться в отсутствии ошибок.

```
Python 3.4.3 Shell
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
e was found 1 time(s).
i was found 2 time(s).
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
a was found 1 time(s).
e was found 6 time(s).
i was found 3 time(s).
u was found 1 time(s).
>>>
                                                                Ln: 23 Col: 4
```

Все отлично. Ошибка не появляется.

Использование метода `setdefault` решило проблему возникновения `KeyError` в нашем коде. Этот прием позволяет динамически добавлять в словарь новые пары ключ/значение во время работы программы, когда это действительно необходимо.

При использовании `setdefault` таким способом вам **никогда** не придется тратить время на предварительную инициализацию всех записей в словаре.

Словари: дополним список изученного

Добавим в список изученного все, что мы узнали о словарях в Python.

КОНТРОЛЬНЫЙ СПИСОК

- По умолчанию словари являются неупорядоченными коллекциями, поскольку для них не поддерживается порядок добавления записей. Если нужно отсортировать словарь, можно воспользоваться встроенной функцией `sorted`.
- Метод `items` позволяет организовать итерации по записям в словаре, то есть по парам ключ/значение. В каждой итерации метод `items` возвращает две переменные цикла — ключ и ассоциированное с ним значение.
- При обращении по несуществующему ключу генерируется исключение `KeyError`. Когда возникает ошибка `KeyError`, программа завершается аварийно.
- Вы можете избежать ошибки `KeyError`, гарантировав добавление в словарь каждого ключа с ассоциированным значением, прежде чем пытаться получить доступ по этому ключу. Здесь могут помочь операторы `in` и `not in`, но чаще используется прием с методом `setdefault`.

Разве словарей (и списков) недостаточно?

Мы уже целую вечность говорим про структуры данных... сколько еще это будет продолжаться? Наверняка словарей вместе со списками мне будет достаточно в большинстве случаев?

Словари (и списки) замечательны.

Но это не единственные структуры данных.

Конечно, большинство задач можно решить с помощью словарей и списков, и программистам на Python редко нужны еще какие-то структуры данных. Но, по правде говоря, эти программисты упускают из виду, что есть еще две структуры данных — **множества** и **кортежи**. Они полезны в некоторых *особых случаях*, и иногда их использование очень упрощает код.

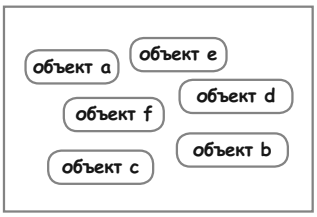
Главная проблема — *идентифицировать* эти особые условия. Чтобы помочь в этом, рассмотрим типичные примеры использования множеств и кортежей и начнем с множеств.

это не Глупые вопросы

В: И со словарями все? А разве для словарей не является обычной практикой, например, сохранять в виде значений списки или другие словари?

О: Да, так обычно и бывает. В конце главы мы покажем, как это делается. А пока вы знаете о словарях достаточно…

Множества не допускают повторений



Множество

В Python **множество** — это структура данных, напоминающая те множества, которые вы изучали в школе: у них есть определенные математические свойства, которые всегда соблюдаются, а главной особенностью является *автоматическое удаление повторяющихся значений*.

Представьте, что есть длинный список имен сотрудников в большой организации, но вас интересует (более короткий) список уникальных имен. Нужен быстрый способ удалить все повторения из первоначального списка. Множества очень удобны для этого: просто преобразуйте длинный список имен в множество (это удалит повторения), а затем — снова в список и — та-дам! — у вас есть список уникальных имен.

Множества оптимизированы для быстрого поиска; работать с ними быстрее, чем со списками, когда главным требованием является поиск. В списках реализован медленный, последовательный алгоритм поиска, поэтому для реализации задач поиска предпочтительнее использовать множества.

Обозначение множеств в коде

Множества легко узнать в коде: коллекция объектов, отделенных друг от друга запятыми и обрамленных фигурными скобками.

Например, вот множество гласных.

Множество начинается и заканчивается фигурными скобками.

```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

Обратите внимание на порядок. Он отличается от порядка ввода. Также удалены повторения.

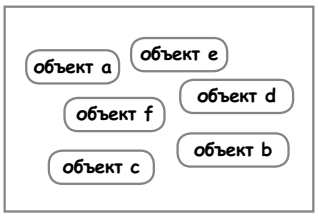
Объекты отделяются друг от друга запятыми.

Необходимость выделять множества фигурными скобками может приводить к путанице между множествами и словарями, которые *тоже* заключаются в фигурные скобки. Главное различие — в использовании двоеточия `(:)` в словарях для отделения ключей и значений друг от друга. В множествах никогда не бывает двоеточий — только запятые.

Кроме запрета на повторения, множества — как и словари — *не* поддерживают порядок добавления элементов. Однако, как и в случае с другими структурами данных, информацию из множеств можно выводить в упорядоченном виде с помощью функции `sorted`. Как списки и словари, множества позволяют добавлять и удалять элементы по мере необходимости.

Будучи множеством, эта структура данных поддерживает операции над множествами, такие как *разность*, *пересечение* и *объединение*. Для демонстрации обратимся снова к программе подсчета гласных в слове. Мы обещали показать, как использовать множества в программе `vowels3.py`, пора выполнить обещание.

Создаем множества эффективно



Множество

Давайте рассмотрим программу `vowels3.py`, где используется список для запоминания гласных в слове.

Вот этот код. Обратите внимание на логику, которая гарантирует запись каждой найденной гласной в список ровно один раз. То есть мы специально убеждаемся, что гласная не будет повторно добавлена в список `found`.

Это программа «vowels3.py», которая выводит каждую гласную, найденную в слове, ровно один раз. В этом коде в качестве структуры данных используются списки.

vowels3.py - /Users/Paul/Desktop/_NewBook/ch02/vowels3.py (3.4.3)

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Ln: 11 Col: 0

Мы предотвращаем появление повторов в списке «found».

Прежде чем продолжить, сохраните этот код в файле с именем `vowels7.py`, чтобы мы могли вносить изменения, не боясь испортить прошлое решение (которое работает). Как мы уже привыкли, сначала поэкспериментируем в консоли, а потом займемся кодом `vowels7.py`. Мы отредактируем код в окне редактора IDLE, когда получим то, что нам нужно.

Создание множеств из последовательностей

Начнем с создания множества гласных, используя код из середины прошлой страницы (можно пропустить этот шаг, если вы уже вводили код в консоли).

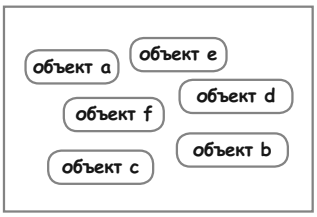
```
>>> vowels = { 'a', 'e', 'e', 'i', 'o', 'u', 'u' }
>>> vowels
{'e', 'u', 'a', 'i', 'o'}
```

Эти две строки кода делают одно и то же: они обе присваивают переменной новый объект множества.

Ниже описан более короткий способ, который позволяет создать множество, просто передав функции `set` последовательность (например, строку). Вот пример создания множества гласных с помощью функции `set`.

```
>>> vowels2 = set('aeeiouu')
>>> vowels2
{'e', 'u', 'a', 'i', 'o'}
```

Использование методов множеств



Множество

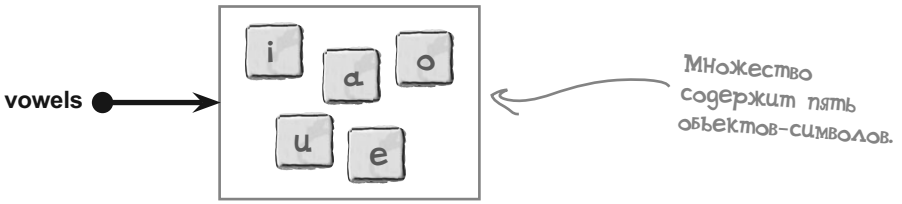
Теперь гласные организованы в виде множества. Следующий шаг — взять слово и выяснить, есть ли среди его букв гласные. Мы можем сделать это, проверив вхождение каждой буквы в множество, поскольку оператор `in` работает с множествами так же, как со словарями и списками. То есть мы можем использовать `in`, чтобы определить, содержит ли множество букву, а затем организовать цикл по всем буквам в слове с помощью `for`.

Однако мы не будем следовать этой стратегии, поскольку для выполнения этой работы мы можем использовать методы множеств.

Есть более эффективные способы выполнения подобных операций при работе с множествами. Мы можем использовать методы, имеющиеся у каждого множества, и выполнять такие операции, как объединение, разность и пересечение. Прежде чем изменить код в `vowels7.py`, поэкспериментируем в консоли и посмотрим, как интерпретатор работает с этими данными. Обязательно выполняйте предлагаемые примеры. Начнем с создания множества гласных, а затем присвоим значение переменной `word`.

```
>>> vowels = set('aeiou')
>>> word = 'hello'
```

Интерпретатор создает два объекта: одно множество и одну строку. Вот так множество `vowels` выглядит в памяти интерпретатора.



Посмотрим, что получится в результате объединения множества гласных и множества букв в переменной `word`. Создадим второе множество, передав «на лету» переменную `word` функции `set`, которое затем передадим методу `union`. В результате получится еще одно множество, которое мы присвоим другой переменной (в данном случае `u`). Эта новая переменная — *комбинация* объектов обоих множеств (объединение).

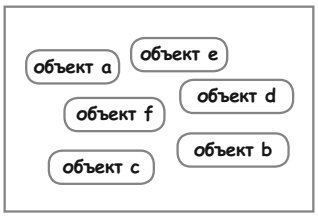
```
>>> u = vowels.union(set(word))
```

Python конвертирует значение переменной «word» в множество объектов-символов (попутно удаляя все повторения).

Метод «union» объединяет два множества, а результат объединения присваивается новой переменной «u» (которая является другим множеством).

Как выглядят множества *vowels* и *u* после вызова метода `union`?

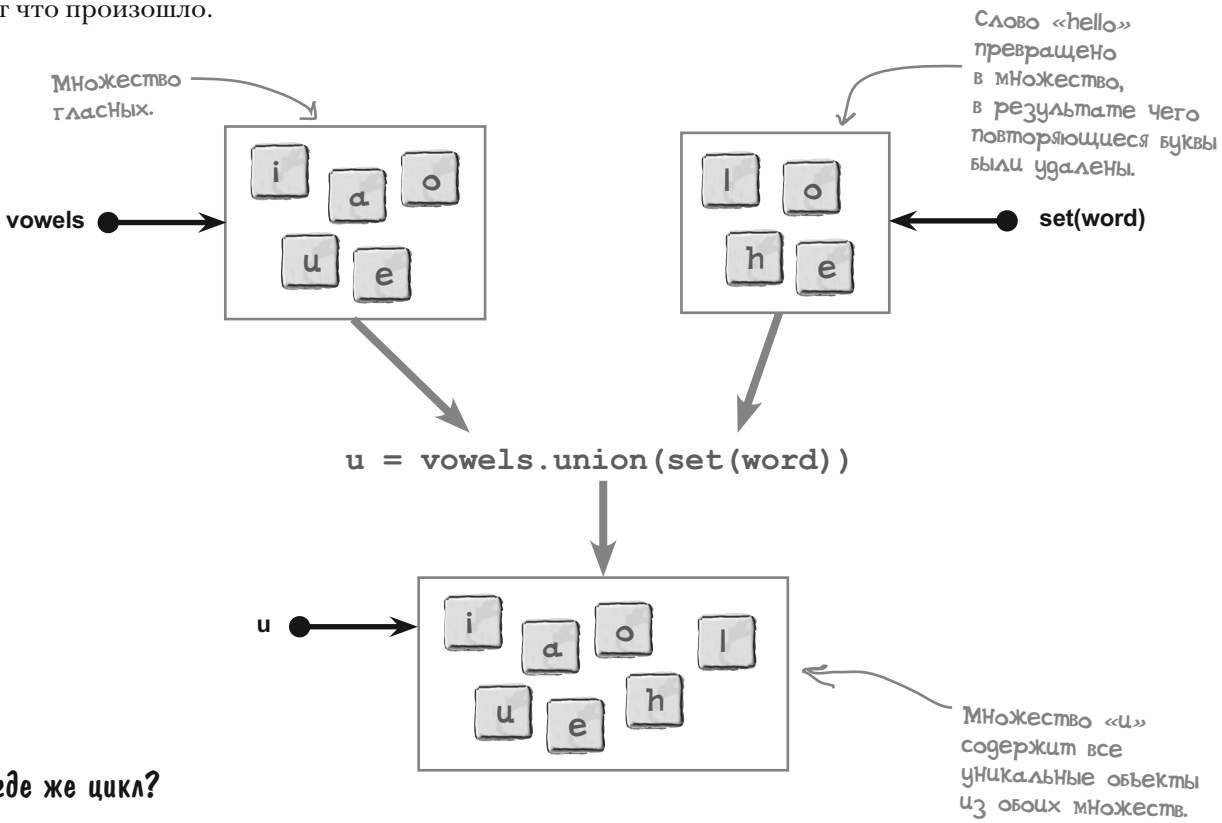
Метод union объединяет множества



Множество

В конце предыдущей страницы мы использовали метод `union` для создания нового множества с именем `u`, объединив множество `vowels` с гласными буквами и множество уникальных букв из переменной `word`. Создание нового множества не повлияло на `vowels`, после объединения множество не изменилось. Однако `u` — новое множество, оно является результатом выполнения метода `union`.

Вот что произошло.



А где же цикл?

Эта отдельная строка кода таит в себе большую силу. Обратите внимание, что вам не пришлось специально инструктировать интерпретатор, чтобы выполнить цикл. Вместо этого вы просто сообщили интерпретатору, *что* вы хотите сделать (а не *как* это сделать), и интерпретатор взял на себя всю работу по созданию нового множества объектов.

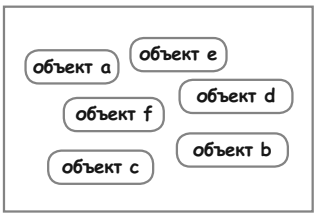
Нередко после создания объединения требуется превратить полученное множество в отсортированный список. Сделать это очень легко, благодаря функциям `sorted` и `list`.

Отсортированный список букв без повторений.

```
>>> u_list = sorted(list(u))
>>> u_list
['a', 'e', 'h', 'i', 'l', 'o', 'u']
```

Метод difference подскажет различия

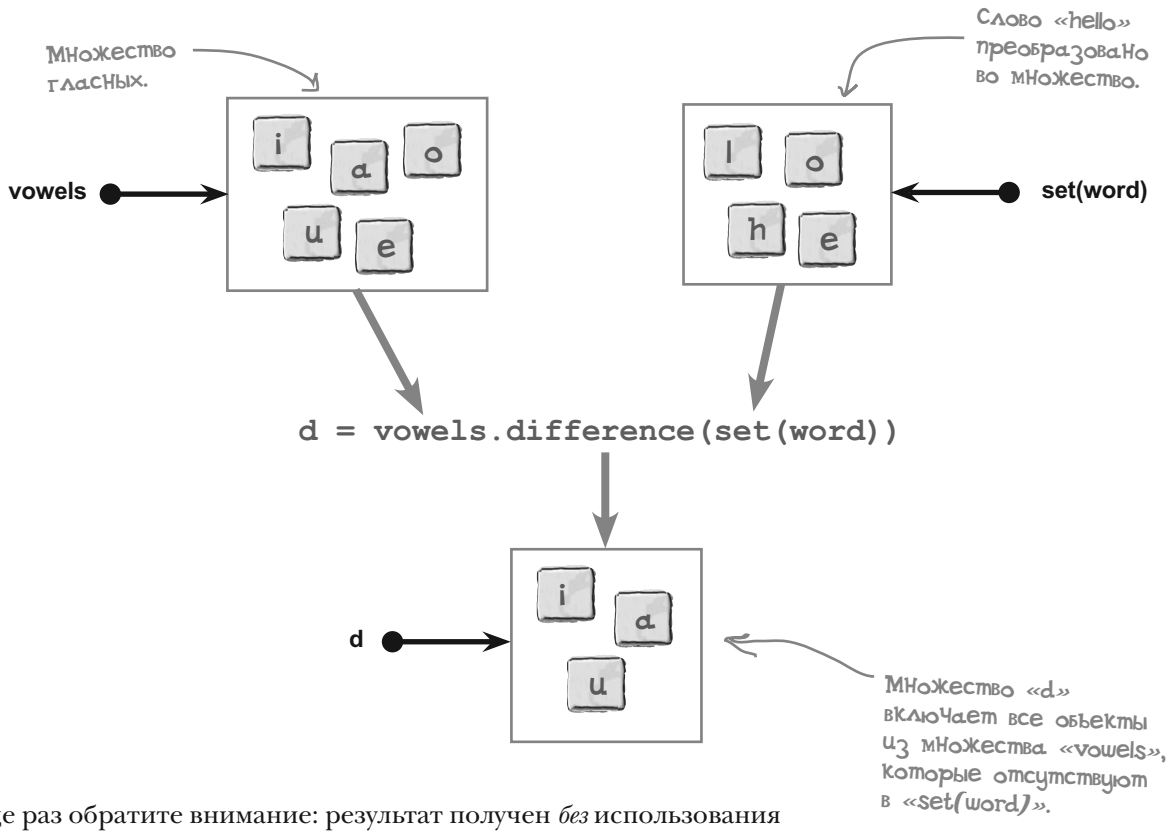
Еще один метод для работы с множествами — `difference` — сообщает, какие объекты присутствуют в одном, но отсутствуют в другом множестве. Давайте проделаем с методом `difference` тот же эксперимент, что и с `union`, и посмотрим, что получится.



Множество

```
>>> d = vowels.difference(set(word))
>>> d
{'u', 'i', 'a'}
```

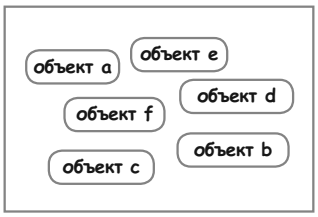
Функция `difference` сравнивает объекты из множества `vowels` с объектами из множества `set(word)`, затем возвращает новое множество объектов (мы назвали его `d`), которые содержатся в `vowels` и *не* содержатся в `set(word)`.



Еще раз обратите внимание: результат получен *без* использования цикла `for`. Функция `difference` выполняет всю работу самостоятельно; мы только передали необходимые данные.

Переверните страницу, чтобы ознакомиться с методом `intersection`.

Метод intersection выделяет общие объекты



Множество

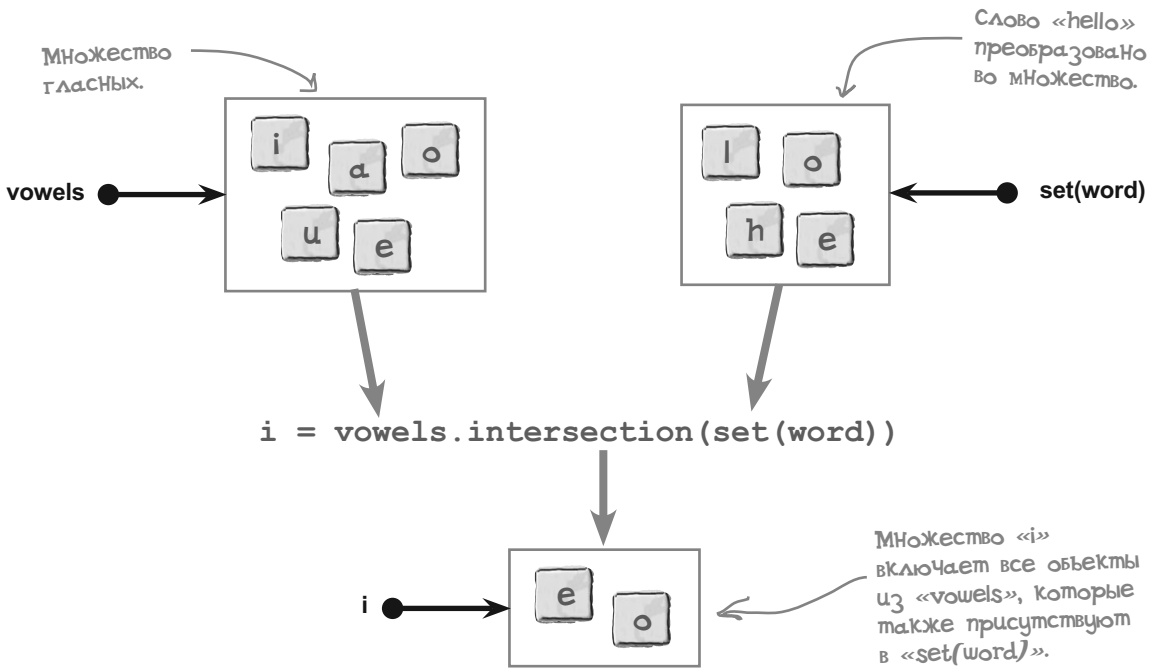
Третий метод, который мы рассмотрим, — это `intersection`. Метод принимает объекты из одного множества и сравнивает их с объектами из другого, а затем сообщает об общих найденных объектах.

С точки зрения программы `vowels7.py` метод `intersection` кажется многообещающим, так как нам нужно узнать, какие из букв в слове — гласные.

Вспомним, что у нас есть строка «`hello`», сохраненная в переменной `word`, и наши гласные в множестве `vowels`. Вот как работает метод `intersection`.

```
>>> i = vowels.intersection(set(word))
>>> i
{'e', 'o'}
```

Метод `intersection` сообщил нам, что в переменной `word` содержатся гласные `e` и `o`. Вот что произошло.



Существует еще много других методов для работы с множествами, кроме тех, что мы рассмотрели на последних нескольких страницах, но из этих трех больше всего нас интересует `intersection`. С помощью одной строки кода мы решили задачу, поставленную в начале прошлой главы: *найти в строке все гласные буквы*. И нам не пришлось использовать цикл. Вернемся к программе `vowels7.py` и применим новое знание.

Множества: что мы уже знаем

Вот краткий обзор того, что мы уже узнали о множествах в Python.

КОНТРОЛЬНЫЙ СПИСОК

- Множества в Python не допускают повторений.
- Как и словари, множества заключаются в фигурные скобки, но в множествах нет пар ключ/значение. Вместо этого каждый уникальный объект в множестве отделяется от следующего запятой.
- Множества, как и словари, не поддерживают порядок добавления (но их можно упорядочивать с помощью функции `sorted`).
- В функцию `set` можно передать любую последовательность объектов, чтобы создать множество элементов из объектов последовательности (минус любые повторения).
- Для работы с множествами есть много встроенных функций, включая методы, выполняющие такие операции, как объединение, разность и пересечение.

Заточите карандаш

Ниже еще раз приводится код программы `vowels3.py`.

Опираясь на новые знания о множествах, возьмите карандаш и вычеркните ненужный код. В пустые строки справа впишите новый код, который позволит воспользоваться преимуществами множеств.

Подсказка: кода получится намного меньше.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

```
word = set(word)
found = word.intersection(set(vowels))
```

Когда все сделаете, сохраните файл с именем `vowels7.py`.

Заточите карандаш
Решение

Ниже еще раз приводится код программы `vowels3.py`.

Опираясь на новые знания, вы должны были взять карандаш и вычеркнуть весь ненужный код. В пустые строки справа вы должны были вписать код, который позволит воспользоваться преимуществами множеств.

Подсказка: кода получится намного меньше.

Довольно много кода нужно вычеркнуть.

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")
found = []
for letter in word:
    if letter in vowels:
        if letter not in found:
            found.append(letter)
for vowel in found:
    print(vowel)
```

Создание множества гласных.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Эти пять строк кода, обрабатывающего списки, заменила единственная строка для работы с множествами.

Когда все сделаете, сохраните файл с именем `vowels7.py`.

Я чувствую себя обманутой… выходит, время на изучение словарей и списков было потрачено впустую и лучшим решением задачи с гласными является использование множеств? Серьезно?

Это не было пустой тратой времени.

Знание, когда лучше использовать одну встроенную структуру данных, а когда — другую, крайне важно (потому что дает уверенность в правильности выбора). А единственный способ научиться этому — поработать со *всеми* структурами данных. Ни одна встроенная структура данных не подходит на все случаи жизни, у каждой есть свои сильные и слабые стороны. Когда вы это поймете, вы будете лучше подготовлены и научитесь выбирать правильные структуры данных на основе конкретных требований вашей задачи.

Пробная поездка

Запустим программу `vowels7.py`, чтобы убедиться, что наша версия программы со множествами работает так, как ожидалось:

Наш последний код.

vowels7.py - /Users/Paul/Desktop/_NewBook/ch03/vowels7.py (3.4.3)

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: hitch-hiker
e
i
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: Galaxy
a
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: life, the universe, and everything
i
a
u
e
>>> ================================ RESTART ================================
>>>
Provide a word to search for vowels: sky
>>>
```

Ln: 23 Col: 4

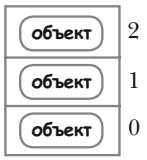
Все работает, как ожидалось.

Множества стали прекрасным выбором...

Но это не говорит о том, что две другие структуры данных не имеют своих сфер применения. Например, для подсчета частоты появления лучше использовать словарь. Однако если важно сохранить порядок добавления, на это способны только списки... и это почти правда. Есть еще одна встроенная структура данных, поддерживающая порядок добавления и пока не обсуждавшаяся. Это **кортежи**.

Остаток главы мы проведем в компании с кортежами в Python.

Сделаем пример с кортежами



Кортеж

Большинство программистов-новичков в Python, когда сталкиваются с **кортежами**, задают вопрос: зачем вообще нужна такая структура данных. В конце концов кортеж — почти как список, который нельзя изменить после создания (и добавить данные). Кортежи неизменяемы: *их нельзя изменить*. Зачем тогда они нужны?

Оказывается, неизменяемые структуры данных часто бывают полезны. Представьте, что вам нужно гарантировать неизменность некоторой структуры данных в программе. Например, у вас есть большой постоянный список (который вы не хотите менять) и вас беспокоит его производительность. Зачем увеличивать стоимость операций со списком, используя дополнительный код для его обработки? Кортежи в таких случаях позволят сократить накладные расходы и гарантировать отсутствие побочных эффектов (где бы они ни появлялись).

Это не Глупые вопросы

В: Как появилось название кортеж (tuple)?

О: Зависит от того, кого вы спрашиваете, но это название пришло из математики. Вы можете прочитать об этом подробнее в *https://ru.wikipedia.org/wiki/Кортеж_(информатика)*.

Как кортежи обозначаются в коде

Поскольку кортежи тесно связаны со списками, неудивительно, что они обозначаются похоже (и ведут себя сходным образом). Кортежи заключены в круглые скобки, а списки — в квадратные. Консоль >>> поможет нам быстро сравнить кортежи и списки. Обратите внимание: использование встроенной функции `type` помогает узнать тип любого созданного объекта.

Ничего нового. Создан список гласных.

```
>>> vowels = [ 'a', 'e', 'i', 'o', 'u' ]
>>> type(vowels)
<class 'list'>
>>> vowels2 = ( 'a', 'e', 'i', 'o', 'u' )
>>> type(vowels2)
<class 'tuple'>
```

Кортеж похож на список, но не является им. Кортежи заключены в круглые (а не в квадратные) скобки.

Встроенная функция «type» сообщает тип объекта.

Теперь, когда `vowels` и `vowels2` созданы (и заполнены данными), попросим интерактивную оболочку вывести их содержимое. Этот шаг подтвердит, что кортежи — не то же самое, что и списки.

```
>>> vowels
['a', 'e', 'i', 'o', 'u']
>>> vowels2
('a', 'e', 'i', 'o', 'u')
```

Круглые скобки показывают, что это кортеж.

Но что произойдет, если попытаться изменить кортеж?

Кортежи неизменяемы

Кортеж

Поскольку кортежи — разновидность списков, они поддерживают операции с квадратными скобками, как и списки. Мы уже знаем, что с помощью квадратных скобок можно получить доступ к содержимому списка. Попробуем заменить букву `i` нижнего регистра в списке `vowels` буквой `I` верхнего регистра.

```
>>> vowels[2] = 'I'
>>> vowels
[' a', 'e', 'I', 'o', 'u']
```

Присвоим букву «I» верхнего регистра третьему элементу списка «vowels».

Как и ожидалось, третий элемент в списке (с индексом `2`) изменился, и это как раз то, что мы хотели увидеть, потому что списки изменяемы. Однако посмотрим, что произойдет, если то же проделать с кортежем `vowels2`.

Интерпретатор громко возмущается в ответ на попытку изменить кортеж.

```
>>> vowels2[2] = 'I'
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    vowels2[2] = 'I'
TypeError: 'tuple' object does not support item assignment
>>> vowels2
(' a', 'e', 'i', 'o', 'u')
```

Ничего не изменилось, потому что кортежи неизменяемы.

Кортежи неизменяемы, поэтому не стоит обижаться на протест компилятора в ответ на попытку изменить объект, хранящийся в кортеже. В конце концов, это отличительная черта кортежа: созданный единожды и заполненный данными, кортеж больше не изменяется.

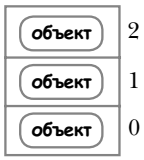
Это поведение очень полезно, особенно если нужно гарантировать неизменность каких-либо данных в программе. Единственный способ сделать это — положить данные в кортеж, и интерпретатор не станет выполнять любой код, пытающийся изменить данные в кортеже.

В оставшейся части книги мы всегда будем использовать кортежи там, где это нужно. В случае с обработкой гласных теперь понятно, почему структуру данных `vowels` следует хранить в кортеже, а не в списке. Нет смысла использовать изменяемую структуру данных (так как пять гласных *никогда* не изменятся).

Нам нечего больше добавить о кортежах — это самые обычные неизменяемые списки. Впрочем, у кортежей есть одна особенность, которая сбивает с толку многих программистов. Давайте посмотрим, что это за особенность и как не попасть из-за нее впросак.

Если данные в вашей структуре никогда не меняются, положите их в кортеж.

Будьте внимательны с кортежами из одного объекта



Кортеж

Представьте, что вы хотите сохранить один строковый объект в кортеже. Есть соблазн поместить строку в скобки, а затем присвоить все это переменной... но это не приведет к ожидаемому результату.

Посмотрим в интерактивной консоли >>>, что произойдет в этом случае.

```
>>> t = ('Python')
>>> type(t)
<class 'str'>
>>> t
'Python'
```

Это не то, чего мы ожидали. У нас в итоге получилась строка. Что же стало с кортежем?

То, что похоже на кортеж из одного объекта, таковым не является; это строка. Таков синтаксический каприз Python. Согласно правилам, чтобы кортеж был кортежем, необходимо включить в его определение хотя бы одну запятую, даже если кортеж содержит один объект. Чтобы создать кортеж с единственным объектом (как мы пытались создать кортеж со строкой в примере выше), нужно включить в определение одну запятую, например так:

```
>>> t2 = ('Python',)
```

Вся разница — в этой запятой в конце, именно по ней интерпретатор узнает, что перед ним кортеж.

Все это кажется немного странным, но пусть вас такое не беспокоит. Просто запомните правило, и все будет отлично: *каждый кортеж должен включать по меньшей мере одну запятую внутри скобок*. Если теперь запросить у интерпретатора вывести тип объекта `t2` (а также его значение), вы узнаете, что `t2` — кортеж, как и ожидалось.

```
>>> type(t2)
<class 'tuple'>
>>> t2
('Python',)
```

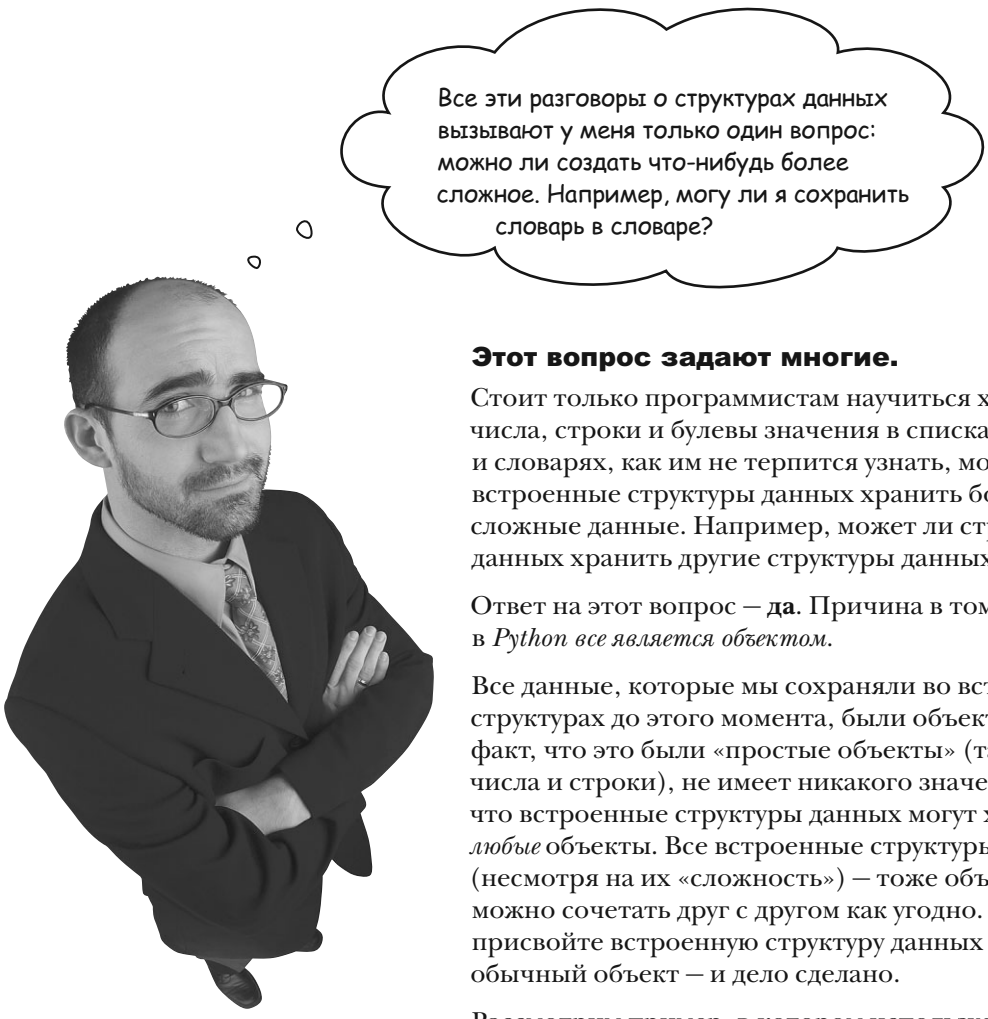
Гораздо лучше: теперь у нас кортеж.

Интерпретатор отображает кортеж с единственным объектом с запятой в конце.

Часто функции принимают и возвращают аргументы в виде кортежа, даже если они принимают или возвращают только один объект. Соответственно, вы часто будете встречать подобный синтаксис, работая с функциями. Мы многое можем рассказать о связи между функциями и кортежами; фактически, этому будет посвящена следующая глава (и вам не придется долго ждать).

Теперь, когда вы познакомились с четырьмя встроенными структурами данных и прежде чем перейти к следующей главе о функциях, давайте сделаем небольшой крюк, чтобы рассмотреть интересный пример с более сложной структурой данных.

Комбинирование встроенных структур данных



Этот вопрос задают многие.

Стоит только программистам научиться хранить числа, строки и булевы значения в списках и словарях, как им не терпится узнать, могут ли встроенные структуры данных хранить более сложные данные. Например, может ли структура данных хранить другие структуры данных?

Ответ на этот вопрос — **да**. Причина в том, что в *Python все является объектом.*

Все данные, которые мы сохраняли во встроенных структурах до этого момента, были объектами. Тот факт, что это были «простые объекты» (такие как числа и строки), не имеет никакого значения, потому что встроенные структуры данных могут хранить *любые* объекты. Все встроенные структуры данных (несмотря на их «сложность») — тоже объекты, и их можно сочетать друг с другом как угодно. Просто присвойте встроенную структуру данных как обычный объект — и дело сделано.

Рассмотрим пример, в котором используется словарь словарей.

Это не Глупые вопросы

В: Это относится только к работе со словарями? Могу я создать список списков, или множество списков, или кортеж словарей?

О: Да, конечно. Мы продемонстрируем, как работает словарь словарей, но вы можете сочетать структуры данных по своему вкусу.

Хранение таблицы данных

Поскольку все является объектом, а все встроенные структуры могут храниться в любой встроенной структуре, позволяя создавать более сложные структуры данных... вряд ли ваш мозг способен все это представить. Например, *словарь списков, содержащих кортежи, которые содержат множества словарей*, может быть хорошей идеей, а может — и не очень, потому что его сложность зашкаливает.

Из сложных структур данных чаще всего используются словари словарей. Эта структура позволяет создавать *изменяемые таблицы*. Представьте, что у нас есть таблица, описывающая коллекцию персонажей.

Имя	Пол	Род занятий	Планета
Форд Префект	Мужской	Исследователь	Бетельгейзе 7
Артур Дент	Мужской	Приготовление бутербродов	Земля
Триша МакМиллан	Женский	Математик	Земля
Марвин	Неизвестен	Робот-параноик	Неизвестна

Вспомните, как в начале этой главы мы создали словарь `person3`, чтобы сохранить данные о Форде Префекте.

```
person3 = { 'Name': 'Ford Prefect',
            'Gender': 'Male',
            'Occupation': 'Researcher',
            'Home Planet': 'Betelgeuse Seven' }
```

Чтобы не создавать четыре разных словаря для каждой строки таблицы (и потом пытаться связать их воедино), давайте создадим один словарь и назовем его `people`. Затем будем использовать `people` для хранения произвольного числа других словарей.

Сначала создадим пустой словарь `people`, а потом сохраним в нем данные Форда Префекта по ключу.

Сначала создадим новый, пустой словарь.

```
>>> people = {}
>>> people['Ford'] = { 'Name': 'Ford Prefect',
                       'Gender': 'Male',
                       'Occupation': 'Researcher',
                       'Home Planet': 'Betelgeuse Seven' }
```

Ключ — «Ford», а значение — другой словарь.

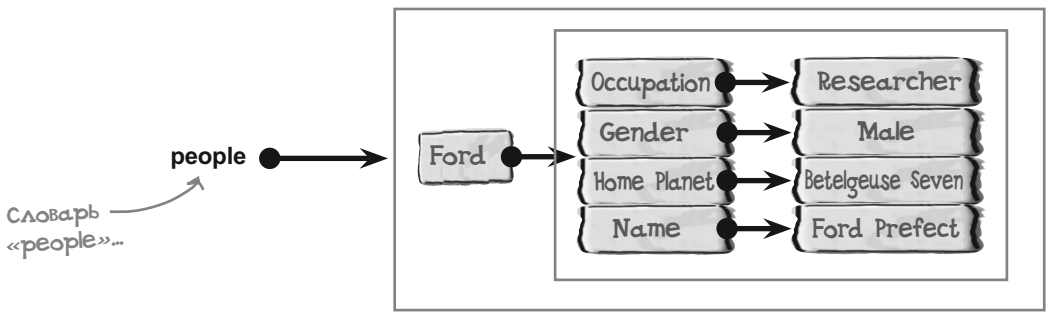
Словарь, содержащий словарь

Словарь встроен в словарь — обратите внимание на дополнительные фигурные скобки.

Создав словарь `people` и добавив в него одну запись с данными (о Форде), можно запросить у интерпретатора вывести содержимое словаря `people` в командной строке >>>. Результат немного удивляет, но все наши данные здесь.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'}}
```

Сейчас в словаре `people` есть только один вложенный словарь, поэтому называть его «словарь словарей» — преувеличение, ведь `people` содержит только один словарь. Вот так `people` выглядит с точки зрения интерпретатора.



Словарь «people»...

...содержит другой словарь (в виде значения, ассоциированного с ключом «Ford»).

Продолжим и добавим остальные три записи в нашу таблицу.

Данные об Артуре.

```
>>> people['Arthur'] = { 'Name': 'Arthur Dent',
                         'Gender': 'Male',
                         'Occupation': 'Sandwich-Maker',
                         'Home Planet': 'Earth' }
>>> people['Trillian'] = { 'Name': 'Tricia McMillan',
                           'Gender': 'Female',
                           'Occupation': 'Mathematician',
                           'Home Planet': 'Earth' }
>>> people['Robot'] = { 'Name': 'Marvin',
                        'Gender': 'Unknown',
                        'Occupation': 'Paranoid Android',
                        'Home Planet': 'Unknown' }
```

Данные о Трише ассоциированы с ключом «Trillian».

Данные о Марвине ассоциированы с ключом «Robot».

Словарь словарей (он же таблица)

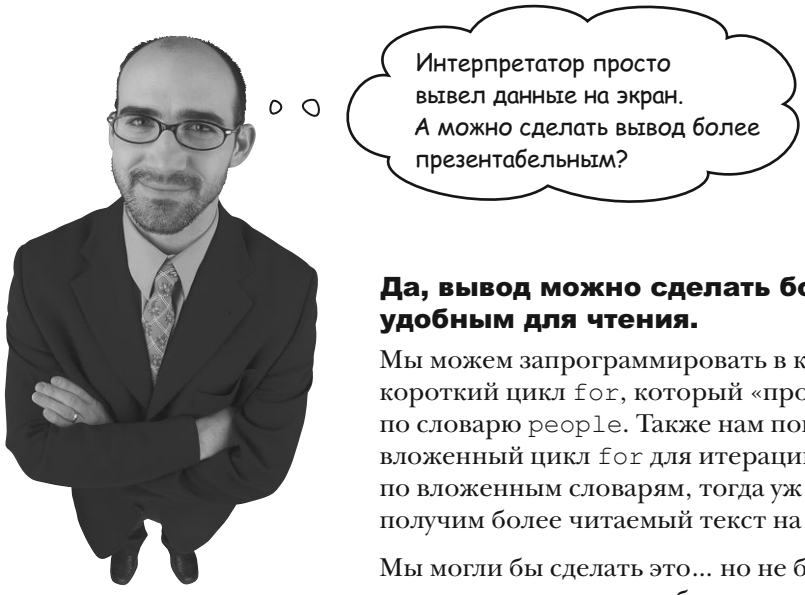
Теперь у нас есть словарь `people` с четырьмя вложенными словарями, и мы можем запросить у интерпретатора вывести содержимое словаря `people` в консоли `>>>`.

В результате на экране появилась какая-то мешанина из данных (смотрите ниже).

Несмотря на мешанину, все наши данные здесь. Обратите внимание: с каждой открывающей фигурной скобки начинается новый словарь, а каждая закрывающая фигурная скобка завершает словарь. Подсчитайте, сколько их (здесь их по пять каждого вида).

Трудновато читать, но все данные здесь.

```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
```



Да, вывод можно сделать более удобным для чтения.

Мы можем запрограммировать в консоли короткий цикл `for`, который «пробежит» по словарю `people`. Также нам понадобится вложенный цикл `for` для итераций по вложенным словарям, тогда уж точно мы получим более читаемый текст на экране.

Мы могли бы сделать это... но не будем, потому что за нас эту работу уже выполнили.

Красивый вывод сложных структур данных

В стандартную библиотеку входит модуль `pprint`, который принимает любую структуру данных и выводит ее в удобном для чтения формате. Название `pprint` означает «pretty print» (красивый вывод).

Давайте используем `pprint` для вывода словаря `people` (словаря словарей). Ниже мы снова вывели данные «в сыром виде» в консоли `>>>`, а затем импортировали модуль `pprint` и использовали его функцию `pprint`, чтобы получить желаемый вывод.

Наш словарь словарей трудно читать.

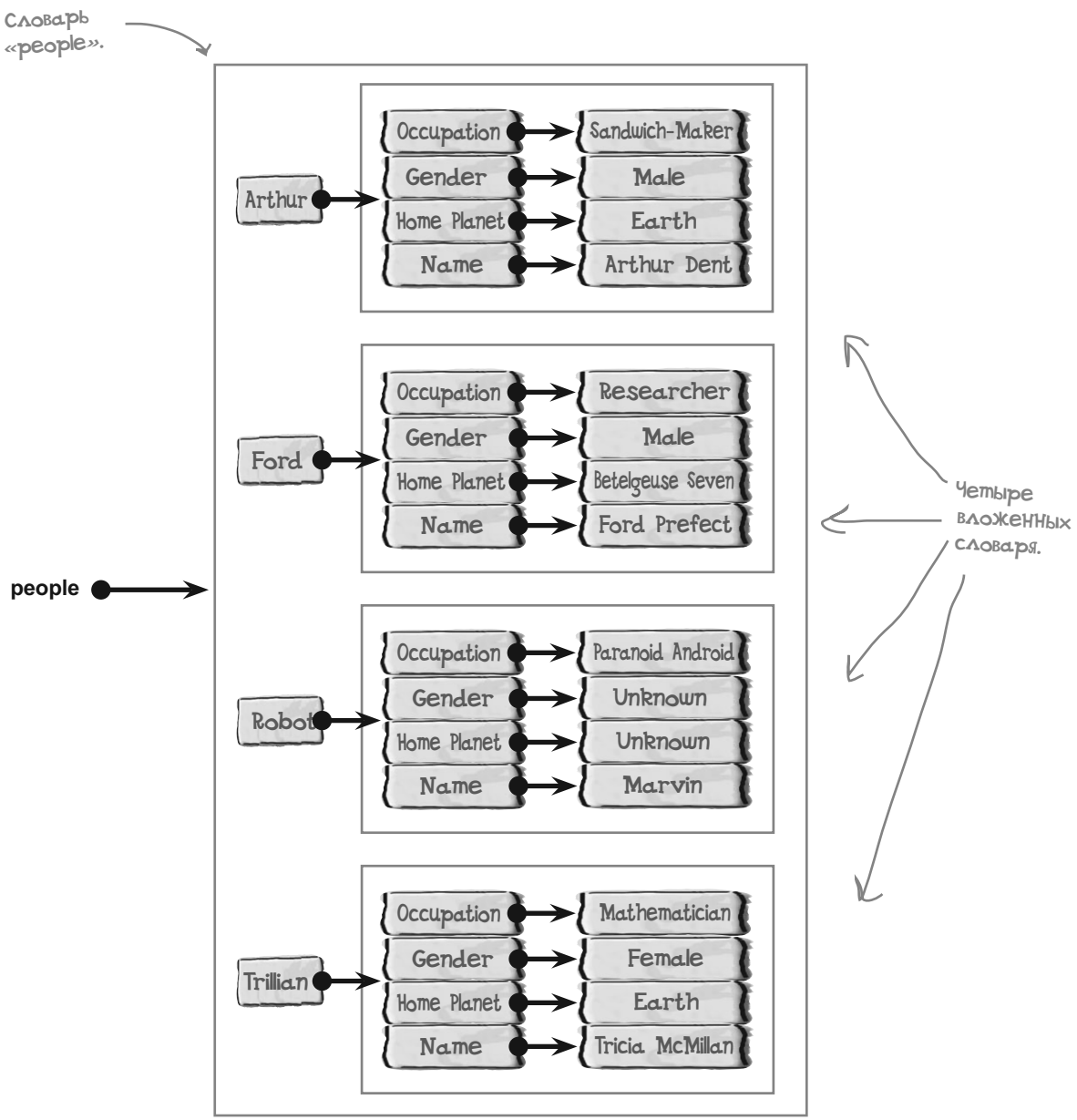
```
>>> people
{'Ford': {'Occupation': 'Researcher', 'Gender': 'Male',
'Home Planet': 'Betelgeuse Seven', 'Name': 'Ford Prefect'},
'Trillian': {'Occupation': 'Mathematician', 'Gender':
'Female', 'Home Planet': 'Earth', 'Name': 'Tricia
McMillan'}, 'Robot': {'Occupation': 'Paranoid Android',
'Gender': 'Unknown', 'Home Planet': 'Unknown', 'Name':
'Marvin'}, 'Arthur': {'Occupation': 'Sandwich-Maker',
'Gender': 'Male', 'Home Planet': 'Earth', 'Name': 'Arthur
Dent'}}
>>>
>>> import pprint
>>>
>>> pprint.pprint(people)
{'Arthur': {'Gender': 'Male',
            'Home Planet': 'Earth',
            'Name': 'Arthur Dent',
            'Occupation': 'Sandwich-Maker'},
 'Ford': {'Gender': 'Male',
          'Home Planet': 'Betelgeuse Seven',
          'Name': 'Ford Prefect',
          'Occupation': 'Researcher'},
 'Robot': {'Gender': 'Unknown',
           'Home Planet': 'Unknown',
           'Name': 'Marvin',
           'Occupation': 'Paranoid Android'},
 'Trillian': {'Gender': 'Female',
              'Home Planet': 'Earth',
              'Name': 'Tricia McMillan',
              'Occupation': 'Mathematician'}}
```

Импорт модуля «pprint», а затем использование функции «pprint».

Такой вывод читать проще. Заметьте: у нас по-прежнему пять открывающих и пять закрывающих фигурных скобок. Однако теперь — благодаря «pprint» — их проще заметить (и сосчитать).

Визуализация сложных структур данных

Обновим представление о том, как интерпретатор «видит» словарь словарей `people` теперь, когда в нем стало больше данных.



Возникает уместный вопрос. *Теперь мы видим, как все эти данные хранятся в словаре словарей, но как мы можем получить их?* Ответ — на следующей странице.

Доступ к данным, хранящимся в сложных структурах

Теперь у нас есть таблица данных, сохраненная в словаре `people`. Напомним, как выглядит эта таблица.

Имя	Пол	Род занятий	Планета
Форд Префект	Мужской	Исследователь	Бетельгейзе 7
Артур Дент	Мужской	Приготовление бутербродов	Земля
Триша МакМиллан	Женский	Математик	Земля
Марвин	Неизвестен	Робот-параноик	Неизвестна

Если нас спросят, чем занимается Артур, мы найдем строку, в которой колонка **Имя** содержит имя Артура, а затем найдем пересечение этой строчки с колонкой **Род занятий**, и тогда мы сможем прочитать «Приготовление бутербродов».

Когда нужно получить доступ к сложной структуре данных (например, как наш словарь словарей `people`), мы можем действовать схожим образом, и сейчас мы продемонстрируем это в командной строке >>>.

Сначала найдем в словаре `people` данные об Артуре, для чего достаточно поместить соответствующий ключ между квадратными скобками.

Найдем запись с данными об Артуре.

```
>>> people['Arthur']
{'Occupation': 'Sandwich-Maker', 'Home Planet': 'Earth',
'Gender': 'Male', 'Name': 'Arthur Dent'}
```

Запись в словаре, ассоциированная с ключом «Arthur».

Мы нашли строку с данными об Артуре и теперь можем найти значение, ассоциированное с ключом `Occupation` (Род занятий). Используем **вторую** пару квадратных скобок для поиска во вложенном словаре и доступа к искомым данным.

Укажем запись.

Укажем столбец.

```
>>> people['Arthur']['Occupation']
'Sandwich-Maker'
```

Использование двойных квадратных скобок позволяет получить доступ к любому значению в таблице по указанным записи и столбцу. Запись соответствует ключу, использованному для включения в словарь (например, `people`), а столбец — ключу во вложенном словаре.

Данные любой сложности

Не важно, какие у вас данные — простой список или что-то более сложное (например, словарь словарей), приятно знать, что встроенные структуры Python способны удовлетворить все потребности. Особенно приятна динамическая природа структур данных; кроме кортежей, каждая из структур данных может расти и сокращаться по мере необходимости, а интерпретатор Python возьмет на себя всю работу по выделению/освобождению памяти.

Мы еще не закончили со структурами данных и вернемся к этой теме позже. Теперь вы знаете достаточно, чтобы продолжить изучение языка.

В следующей главе мы поговорим об эффективных технологиях повторного использования кода в Python и ознакомимся с одним из самых основных элементов этих технологий: функциями.

Код из 3-й главы, 1 из 2

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

found['a'] = 0
found['e'] = 0
found['i'] = 0
found['o'] = 0
found['u'] = 0

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

Программа «vowels4.py» выполняет подсчет частоты гласных. Она основана на программе «vowels3.py», которую мы впервые встретили в главе 2.

Пытаясь удалить инициализацию словаря, мы создали «vowels5.py». Эта программа завершается аварийно с ошибкой времени выполнения (потому что мы не инициализировали счетчики частоты).

```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

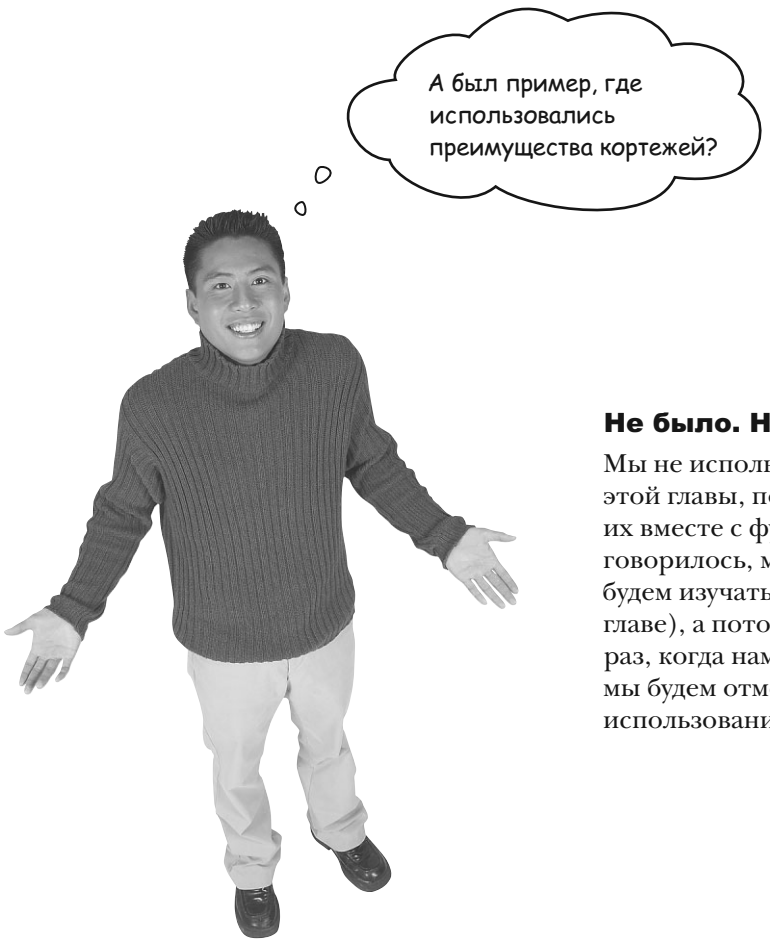
```
vowels = ['a', 'e', 'i', 'o', 'u']
word = input("Provide a word to search for vowels: ")

found = {}

for letter in word:
    if letter in vowels:
        found.setdefault(letter, 0)
        found[letter] += 1

for k, v in sorted(found.items()):
    print(k, 'was found', v, 'time(s).')
```

В программе «vowels6.py» исправлена ошибка с помощью метода «setdefault», который может работать с любым словарем (и присваивать ключу значение по умолчанию, если в словаре ключ отсутствует).

Код из 3-й главы, 2 из 2

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

В последней версии программы для работы с гласными «vowels7.py» используются преимущества множеств. Таким образом мы существенно сократили программу «vowels3.py», основанную на списках, и сохранили ту же функциональность.



Не было. Но это нормально.

Мы не использовали кортежи в примерах этой главы, потому что позже рассмотрим их вместе с функциями. Как уже говорилось, мы встретим кортежи, когда будем изучать функции (в следующей главе), а потом и в других главах. Каждый раз, когда нам встретятся кортежи, мы будем отмечать особенности их использования.

4 повторное использование

Функции и модули

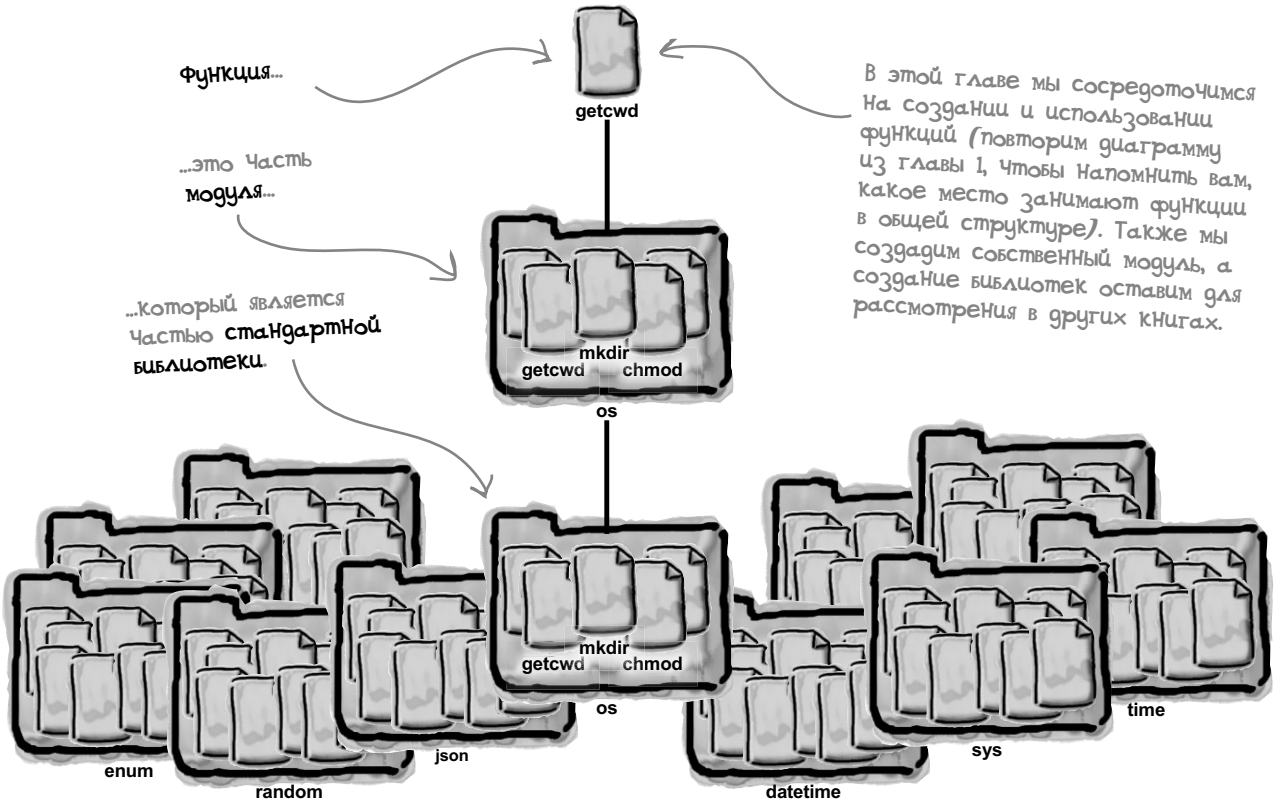
Повторное использование кода — ключ к построению стабильных систем.

В случае Python все повторное использование начинается и заканчивается **функциями**. Возьмите несколько строк кода, дайте им имя — и у вас готовая функция (которую можно использовать повторно). Возьмите коллекцию функций и сохраните их в файле — у вас готовый **модуль** (который можно использовать повторно). Это правда, когда говорят, что *делиться приятно*, и в конце главы вы уже сможете создавать код для **многократного** и **совместного** использования, благодаря пониманию того, как работают функции и модули Python.

Повторное использование кода с помощью функций

Несколько строк кода на Python могут выполнить довольно много работы, но рано или поздно вы обнаружите, что кода стало слишком много... и его все труднее сопровождать. Что будет, если 20 строк кода на Python разрастутся до 500 строк или более? В этом случае пора задуматься о стратегиях, позволяющих уменьшить сложность программы.

Как и многие языки программирования, Python поддерживает **модульность**, позволяя разбивать большие фрагменты кода на меньшие, более удобные для сопровождения. Для этого создаются **функции**, которые проще представить как имена для фрагментов кода. Вспомните диаграмму из главы 1, которая иллюстрирует отношения между функциями, модулями и стандартной библиотекой.



В этой главе основное внимание уделяется созданию функций — эта часть показана в верхушке диаграммы. Когда вы освоите создание функций, мы покажем, как создать модуль.

Представляем функции

Прежде чем превратить несколько строк кода в функцию, остановимся на минуту и оценим «анатомию» *любой* функции в Python. После завершения вводной части выберем фрагмент написанного кода и выполним несколько шагов, чтобы превратить его в функцию, пригодную для повторного использования.

Пока не будем вдаваться в детали. Сейчас от вас требуется только понять, как выглядят функции в Python, о чем и рассказывается на этой и следующей страницах. Мы рассмотрим детали по ходу изложения. В окне IDLE на этой странице показан шаблон, который можно использовать для создания любой функции. Пока вы смотрите на него, подумайте вот о чем.

1. **В функциях используются два новых ключевых слова: `def` и `return`.**
Оба выделяются в IDLE оранжевым. Ключевое слово `def` присваивает имя функции (в IDLE выделяется голубым) и определяет ее аргументы. Ключевое слово `return` можно не использовать, оно возвращает значение в код, который вызвал функцию.

2. **Функции могут принимать аргументы.**
Функция может принимать аргументы (то есть входные значения). Список аргументов заключается в круглые скобки и следует в строке с `def` сразу за именем функции.

3. **Функции содержат код и (обычно) документацию.**
Код функции под строкой `def` оформляется дополнительным отступом и может содержать комментарии, если нужно. Мы покажем два способа добавления комментариев: с использованием тройных кавычек (в редакторе IDLE выделяется зеленым и называется **строкой документации**) и с использованием однострочного комментария, который начинается сразу после символа # (в IDLE выделяется красным).

Инструкция «def» определяет имя функции и список ее аргументов.

Строка документации описывает назначение функции.

Удобный шаблон функции.

Поместите свой код здесь (вместо однострочных комментариев).

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """Строка документации."""
    # Здесь находится код вашей функции.
    # Здесь находится код вашей функции.
    # Здесь находится код вашей функции.
    return optional_value
```

Ln: 8 Col: 0

Для умников

Для описания многократно используемых фрагментов кода в Python используется термин «функция». В других языках программирования используются такие понятия, как «процедура», «метод» и «подпрограмма». Если функция является частью класса Python, ее называют «методом». Позже мы расскажем о классах и методах Python.

Как быть с информацией о типах?

Еще раз взгляните на шаблон функции. Пока мы не начали выполнять код, вам не кажется, что чего-то не хватает? Может быть, здесь должно быть указано что-то еще, но мы не сделали этого? Посмотрите внимательно.

function_template.py - /Users/Paul/Desktop/_NewBook/ch04/function_template.py (3.4.3)

```
def a_descriptive_name(optional_arguments):
    """Строка документации."""
    # Здесь находится код вашей функции.
    # Здесь находится код вашей функции.
    # Здесь находится код вашей функции.
    return optional_value
```

Ln: 8 Col: 0

В этом шаблоне чего-то не хватает?

Я слегка в шоке от этого шаблона функции. Как интерпретатор узнает, какие в ней типы аргументов и возвращаемого значения?

Он и не узнает, но не беспокойтесь об этом.

Интерпретатор Python не требует указывать типы аргументов или возвращаемого значения. В зависимости от языка программирования, который вы использовали раньше, это может удивить. Но не беспокойтесь.

Python позволяет передать в качестве аргумента любой *объект*, возвращаемое значение тоже может быть любым *объектом*. Интерпретатор не проверяет типы этих объектов (только факт их передачи).

В Python 3 можно *указать* ожидаемые типы аргументов/возвращаемых значений, и позже мы это сделаем. Однако указание типов не включает как по волшебству проверку типов, потому что Python *никогда* не проверяет типы аргументов и возвращаемых значений.

Назовем фрагмент кода с помощью «def»

Если фрагмент кода для многократного использования выбран, можно приступать к созданию функции. Функция создается с помощью ключевого слова `def` (от англ. *define* — *определение*). За ним следуют имя функции, список аргументов (в круглых скобках), который может быть пустым, двоеточие и тело функции (одна или несколько строк кода).

Вспомните программу `vowels7.py` из прошлой главы, которая выводила гласные, найденные в указанном слове.

Это программа «vowels7.py» из главы 3.

Взять множество гласных...

...и слово...

...затем найти пересечение.

```
vowels = set('aeiou')
word = input("Provide a word to search for vowels: ")
found = vowels.intersection(set(word))
for vowel in found:
    print(vowel)
```

Вывести результат.

Представьте, что вы планируете многократно использовать эти пять строк в более крупной программе. И меньше всего вам хотелось бы копировать и вставлять этот код в нужных местах... поэтому, чтобы программа оставалась простой и содержала только **одну копию** кода, создадим функцию.

Потратьте время, чтобы выбрать для функции хорошее имя, описывающее ее назначение.

Покажем, как это сделать в консоли Python. Чтобы превратить эти пять строк кода в функцию, используйте ключевое слово `def`, начинающее определение функции; дайте функции имя, которое в полной мере отражает ее назначение (это *всегда* хорошая идея); добавьте пустой список аргументов в круглых скобках, после них поставьте двоеточие; а затем добавьте строки кода с отступом относительно `def`, как показано ниже.

Начните с ключевого слова «def».

Дайте функции хорошее описательное имя.

Добавьте список аргументов. В примере у функции нет аргументов, поэтому список пустой.

Не забудьте поставить двоеточие.

```
>>> def search4vowels():
        vowels = set('aeiou')
        word = input("Provide a word to search for vowels: ")
        found = vowels.intersection(set(word))
        for vowel in found:
            print(vowel)
```

Пять строк кода из «vowels7.py» с отступами.

Поскольку код набран в консоли, не забудьте ДВАЖДЫ нажать Enter, чтобы подтвердить завершение ввода кода с отступом.

Теперь, когда функция создана, вызовем ее и убедимся, что она работает так, как мы предполагали.

Вызываем функции

Чтобы вызвать функцию в Python, необходимо набрать имя функции, дополнив его значениями аргументов. Функция `seach4vowels` (сейчас) не имеет аргументов, и мы можем вызвать ее с пустым списком аргументов.

```
>>> search4vowels()
Provide a word to search for vowels: hitch-hiker
e
i
```

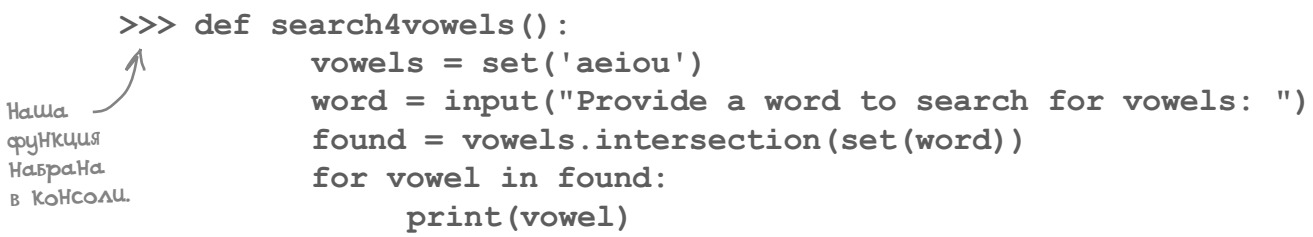
Если вызвать функцию еще раз, она выполнится повторно.

```
>>> search4vowels()
Provide a word to search for vowels: galaxy
a
```

Никаких сюрпризов: вызов функции приводит к выполнению кода.

Редактируем функцию в редакторе, а не в командной строке

Сейчас код функции `search4vowels` набран в консоли >>> и выглядит так:



Чтобы продолжить работу над кодом, можно повторно вызвать его в консоли (например, комбинацией Alt-P в *Windows* или *Linux* или Ctrl-P в *Mac OS X*) и отредактировать, но это не очень удобно. Вспомните: если код, с которым вы работали в консоли >>>, содержит больше пары строк, его лучше скопировать в редактор IDLE, где заниматься редактированием намного удобнее. Сделаем это и продолжим.

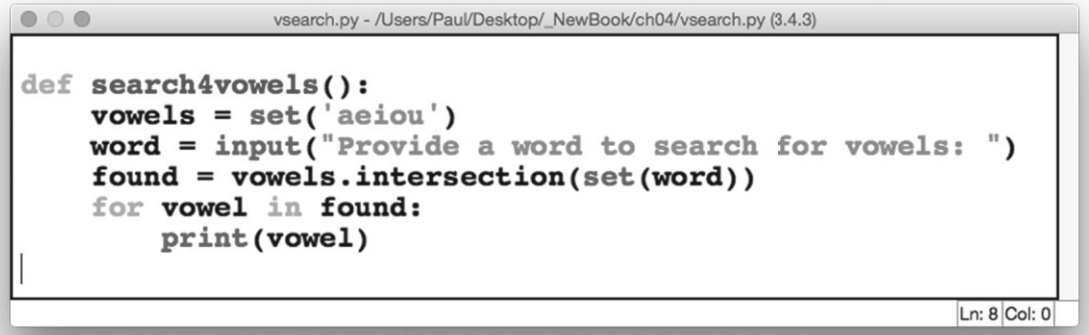
Откройте новое, пустое окно редактора IDLE, скопируйте код функции из консоли (только *не* копируйте символы >>>) и вставьте его в окно редактора. Поправив форматирование и отступы, сохраните файл с именем `vsearch.py`.

Не забудьте сохранить код в файле с именем «vsearch.py» после копирования функции из консоли.

Используем редактор IDLE, чтобы внести изменения

Код функции теперь в редакторе IDLE и сохранен в файле с именем «vsearch.py».

Вот так выглядит файл `vsearch.py` в окне редактора IDLE:



После нажатия F5 происходит следующее: окно с оболочкой IDLE выходит на передний план и производится перезапуск оболочки. Однако на экране ничего не появилось. Попробуйте проделать то же самое, чтобы увидеть, что мы имеем в виду: нажмите F5.

Если IDLE сообщит об ошибке, когда вы нажмете F5, не паникуйте! Вернитесь в окно редактора, проверьте, совпадает ли ваш код с нашим, а потом запустите его снова.

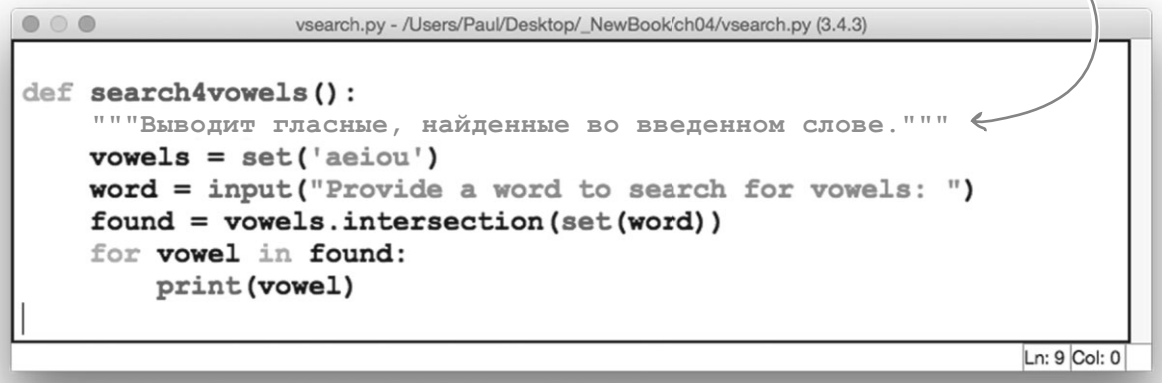
Причина, по которой ничего не выводится на экран, заключается в том, что вы пока не вызвали функцию. Мы вызовем ее немного позже, а сейчас внесем в функцию одно изменение. Небольшое, но крайне важное изменение.

Добавим немного информации в начало функции.

Чтобы добавить многострочный комментарий (**строку документации**) в любой код, заключите текст комментария в тройные кавычки.

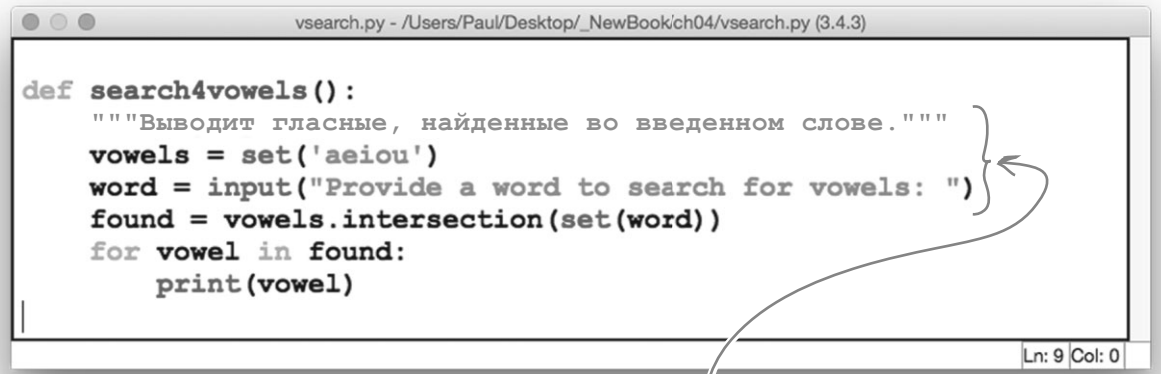
Строка документации добавлена в код функции. Данный комментарий описывает (кратко) назначение функции.

Вот так выглядит файл `vsearch.py` после добавления строки документации. Внесите такие же изменения в свой код.



Почему эти строки оформлены по-разному?

Еще раз взгляните на функцию в ее нынешнем состоянии. Обратите особое внимание на три строки кода, выделенные в редакторе IDLE зеленым:



Подсветка синтаксиса в IDLE показывает, что у нас проблема с единообразием в использовании кавычек. Когда и какой стиль следует использовать?

Выбор кавычек для оформления строк

В Python строки могут заключаться в одиночные (`'`), двойные (`"`) или так называемые тройные (`"""` или `'''`) кавычки.

Как отмечалось выше, строки в тройных кавычках называются **строками документации**, потому что они используются для документирования назначения функций (как показано в примере). Вы можете использовать `"""` или `'''` для обрамления строк документирования, но большинство программистов на Python предпочитают использовать `"""`. Строки документации обладают интересными свойствами — они могут занимать несколько строк (в других языках программирования для обозначения подобных строк используется термин «heredoc» — «встроенный документ»).

Строковые литералы, заключенные в одиночные (`'`) или двойные кавычки (`"`), **не могут** занимать несколько строк: строковый литерал должен завершаться соответствующей кавычкой без переноса на другую строку (в Python конец строки завершает инструкцию).

Какой символ использовать для обрамления строк — решать вам, хотя большинство программистов на Python предпочитают одиночные кавычки. Главное, придерживаться одного выбранного стиля.

Код, показанный в начале страницы (хоть и вполне работоспособный), использует кавычки *не* единообразно. Обратите внимание, что код работает правильно (интерпретатору все равно, какой стиль вы используете), но смешивание стилей делает код более сложным для понимания (и это плохо).

Будьте последовательны в использовании кавычек для обрамления строк. По возможности всегда используйте одиночные кавычки.

Следуйте лучшим рекомендациям, изложенным в PEP

Сообщество программистов на Python потратило много времени, продумывая и документируя рекомендации по оформлению кода (не только строк). Этот передовой опыт известен как **PEP 8**. PEP — сокращение от «Python Enhancement Proposal» (предложения по развитию Python).

Существует большое количество документов PEP, в них описываются детали предложенных и реализованных дополнений для языка Python, но также встречаются советы (что следует делать, а чего не следует) и описываются различные процессы Python. Документы PEP — техническая документация, местами (часто) эзотерического характера. Поэтому большинство программистов на Python знают об их существовании, но редко вникают в детали. Такая ситуация наблюдается с большинством документов PEP, *кроме* PEP 8.

Здесь вы найдете список рекомендаций PEP: https://www.python.org/dev/peps/.

PEP 8 — это руководство по стилю оформления кода на языке Python. Документ *рекомендуется* к прочтению всем программистам на Python, и один из главных советов, который встречается на его страницах: быть последовательным в использовании кавычек для обрамления строк. Потратьте время и прочтите PEP 8 хотя бы раз (на русском языке документ доступен по адресу: http://pep8.ru/doc/pep8/). Еще один документ, PEP 257, посвящен форматированию строк документации, его тоже стоит прочитать.

Взгляните на функцию `search4vowels`, на этот раз оформленную с учетом рекомендаций PEP 8 и PEP 257. Изменения невелики, но стандартные одиночные кавычки, обрамляющие строки (не строки документирования), делают его более приятным для глаза.

Строка документации в стиле PEP 257.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels():
    """Выводит гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    word = input('Provide a word to search for vowels: ')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Ln: 9 Col: 0

Поскольку PEP 8 рекомендует быть последовательными в использовании кавычек, мы выбрали для обрамления строк одиночные кавычки.

Необязательно писать код, который в *точности* соответствует PEP 8. Например, имя функции `search4vowels` не соответствует пожеланиям документа, потому что в стандарте слова в именах функции отделяются символом подчеркивания: имя `search_for_vowels` точнее соответствовало бы рекомендациям. Заметьте: PEP 8 — набор рекомендаций, а не правил. Вы не должны им слепо подчиняться, только принимать во внимание, и нам нравится имя `search4vowels`.

Стоит сказать, что большинство программистов Python будут благодарны вам за код, написанный в духе PEP 8, потому что такой код легче читать.

Теперь вернемся к функции `search4vowels` и добавим в нее аргументы.

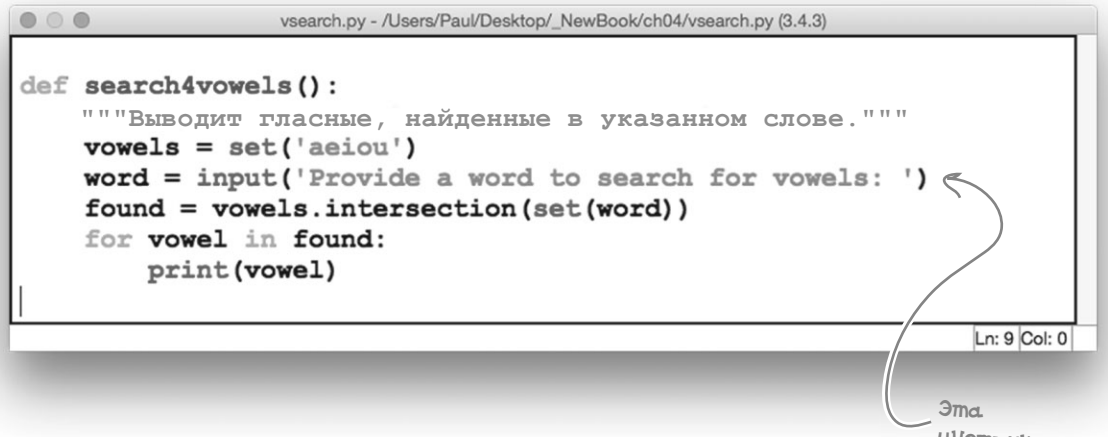
Функции могут принимать аргументы

Прежде чем функция начинает свою работу, пользователь должен ввести слово для анализа. Давайте изменим функцию `search4vowels` так, чтобы она принимала слово в виде входного аргумента.

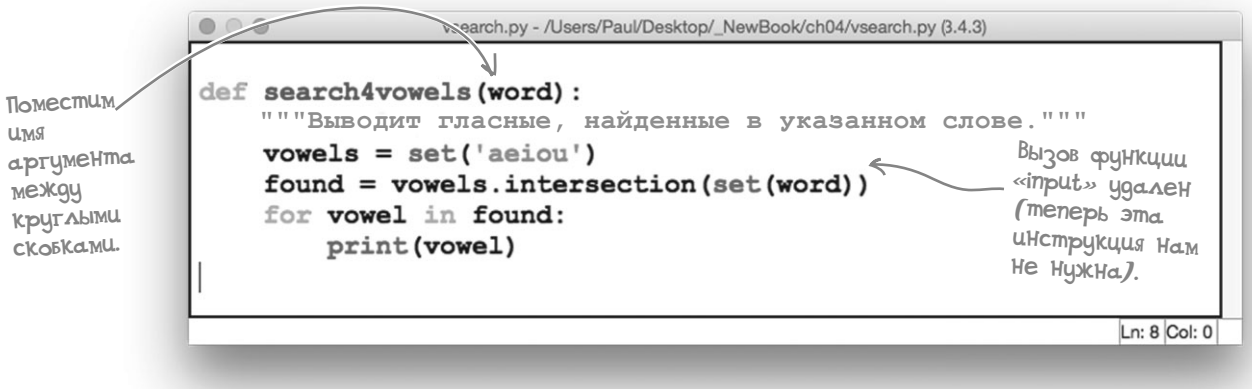
Добавление аргумента — задача простая: вы просто вставляете имя аргумента между круглыми скобками в строке с `def`. Имя аргумента становится переменной в теле функции. Очень просто.

Теперь удалим строку кода, которая запрашивает у пользователя слово для анализа. Это тоже легко сделать.

Напомним, как теперь должен выглядеть код.



После внесения изменений в окне редактирования IDLE должен получиться следующий результат (обратите внимание: мы также изменили строку документирования, поддерживать актуальность которой — *всегда* хорошая идея):



Не забывайте сохранять файл после каждого изменения, прежде чем нажать F5 и выполнить новую версию функции.

Пробная поездка

Поправив код в окне редактора IDLE (и сохранив), нажмите F5, затем несколько раз подряд вызовите функцию и посмотрите, что получится.

Текущая версия «search4vowels».

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(word):
    """Выводит гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Python 3.4.3 Shell

```
>>> ================================ RESTART ================================
>>>
>>> search4vowels()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    search4vowels()
TypeError: search4vowels() отсутствует 1 обязательный позиционный аргумент: 'word'
>>> search4vowels('hitch-hiker')
e
i
>>> search4vowels('hitch-hiker', 'galaxy')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    search4vowels('hitch-hiker', 'galaxy')
TypeError: search4vowels() принимает 1 позиционный аргумент, но было передано 2
>>>
```

Ln: 2 Col: 4

Мы вызвали функцию «search4vowels» три раза, но только один вызов был успешным — когда мы передали в функцию единственный строковый аргумент. Остальные два вызова были неудачными. Прочтите сообщения об ошибках, сгенерированные интерпретатором, чтобы понять, почему некорректные вызовы провалились.

Это не Глупые вопросы

В: **При создании функции в Python можно задавать только один аргумент?**

О: Нет, вы можете задать столько аргументов, сколько пожелаете, все зависит от того, что делает функция. Мы специально начали с простого примера и далее в главе рассмотрим более сложные примеры. В Python можно сделать многое, задавая аргументы функций, и скоро мы обсудим большинство возможностей.

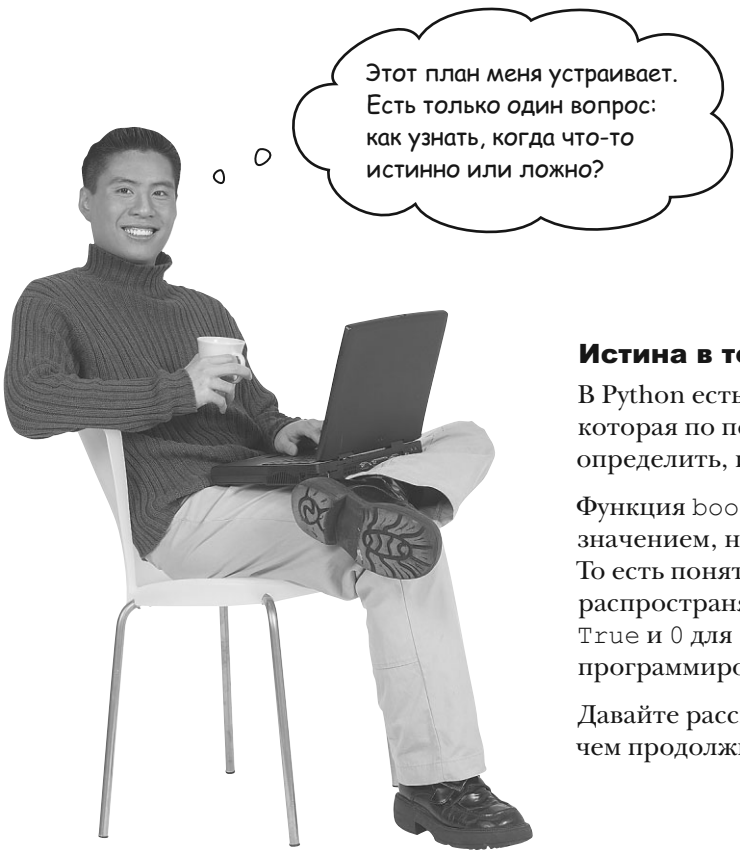
Функции возвращают результат

Обычно программисты создают функции не только для того, чтобы дать имя некоторому фрагменту кода, но и чтобы вернуть вычисленное значение, с которым потом сможет работать вызывающий код. Для возврата значения (или значений) из функции в Python используется выражение `return`.

Когда интерпретатор встречает инструкцию `return` в теле функции, происходит следующее: функция завершает свое выполнение и любое значение, переданное инструкции `return`, возвращается вызывающему коду. Именно так работает инструкция `return` в большинстве других языков программирования.

Начнем с простого примера возврата единичного значения из функции `search4vowels`. В частности, вернем `True` или `False` в зависимости от наличия гласных в слове, переданном в аргументе.

Это *действительно* небольшое отклонение от первоначального назначения нашей функции, однако доверьтесь нам, потому что мы собираемся представить более сложный (и полезный) пример. Начав с простого примера, мы подготовим почву для движения дальше.



Истина в том...

В Python есть встроенная функция `bool`, которая по переданному параметру может определить, истинный он или ложный.

Функция `bool` работает не только с любым значением, но и с любым объектом Python. То есть понятие истинности в Python распространяется не только на `1` для `True` и `0` для `False`, как в других языках программирования.

Давайте рассмотрим `True` и `False`, прежде чем продолжить обсуждение `return`.

Истина под лупой

Каждый объект в Python имеет связанное с ним значение, по которому определяется его истинность или ложность — `True` или `False`.

Объект считается ложным (`False`), если он равен `0`, значению `None`, пустой строке или пустой встроенной структуре данных. Все следующие примеры соответствуют значению `False`:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False
>>> bool(None)
False
```

Если объект равен 0, он всегда False.

Пустая строка, пустой список и пустой словарь оцениваются как False.

Значение «None» в Python всегда False.

Остальные объекты в Python считаются истинными (`True`). Вот примеры истинных объектов.

```
>>> bool(1)
True
>>> bool(-1)
True
>>> bool(42)
True
>>> bool(0.00000000000000000000000000000001)
True
>>> bool('Panic')
True
>>> bool([42, 43, 44])
True
>>> bool({'a': 42, 'b':42})
True
```

Ненулевые числа всегда True, даже отрицательные.

Число может быть очень маленьким, но все же оно не равно 0, поэтому True.

Непустая строка всегда True.

Непустая встроенная структура данных всегда True.

Мы можем передать функции `bool` любой объект и определить его истинность (`True`) или ложность (`False`).

Вообще любая непустая структура данных — это `True`.

Возврат одного значения

Взгляните еще раз на код функции: в данный момент она принимает любое значение в аргументе, ищет в этом значении гласные и выводит найденные гласные на экран.

```
def search4vowels(word):
    """Display any vowels found in a supplied word."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    for vowel in found:
        print(vowel)
```

Мы изменим эти две строки.

Изменить эту функцию, чтобы она возвращала `True` или `False`, в зависимости от присутствия в слове гласных, очень легко. Просто замените последние две инструкции (цикл `for`) этой строкой:

```
return bool(found)
```

Вызов функции «bool» и...

...передача имени структуры данных, которая содержит результат поиска гласных.

Если ничего не найдено, функция вернет `False`; в противном случае — `True`. После внесения изменений можно протестировать новую версию функции в командной строке Python и посмотреть, что произойдет.

```
>>> search4vowels('hitch-hiker')
True
>>> search4vowels('galaxy')
True
>>> search4vowels('sky')
False
```

Инструкция «return» (благодаря функции «bool») дает нам значение «True» или «False».

Как и в предыдущих главах, мы не считаем букву 'y' гласной.

Если вы наблюдаете поведение предыдущей версии функции, убедитесь, что сохранили новую версию, прежде чем нажать F5 в окне редактора.

Для умников

Не поддавайтесь искушению завернуть в скобки выражение, которое возвращает `return`. Не нужно этого делать. Инструкция `return` — это не вызов функции, поэтому скобки не являются синтаксическим правилом. Вы можете их использовать (если *действительно* этого хотите), но большинство программистов на Python этого не делают.

Возврат более одного значения

Функции придуманы для того, чтобы возвращать одно значение, но иногда нужно вернуть несколько значений. Единственный способ сделать это — упаковать значения в структуру данных, а потом вернуть ее. Таким образом, функция все еще будет возвращать единственный объект, но он будет содержать много элементов данных.

Наша функция в ее нынешнем состоянии возвращает булево значение (то есть один объект).

Обратите внимание: мы обновили комментарий.

```
def search4vowels(word):
    """Возвращает булево значение в зависимости
       от присутствия любых гласных."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return bool(found)
```

Мы легко можем изменить функцию, чтобы она возвращала несколько значений (в одном множестве). Достаточно убрать вызов `bool`.

```
def search4vowels(word):
    """Возвращает гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    found = vowels.intersection(set(word))
    return found
```

Мы снова обновили комментарий.

Возврат значений в виде структуры данных (множества).

Мы можем сократить последние две строки кода в новой версии функции до одной, избавившись от ненужной переменной `found`. Вместо присваивания результата функции `intersection` переменной `found` и ее возврата, можно сразу вернуть результат `intersection`.

```
def search4vowels(word):
    """Возвращает гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

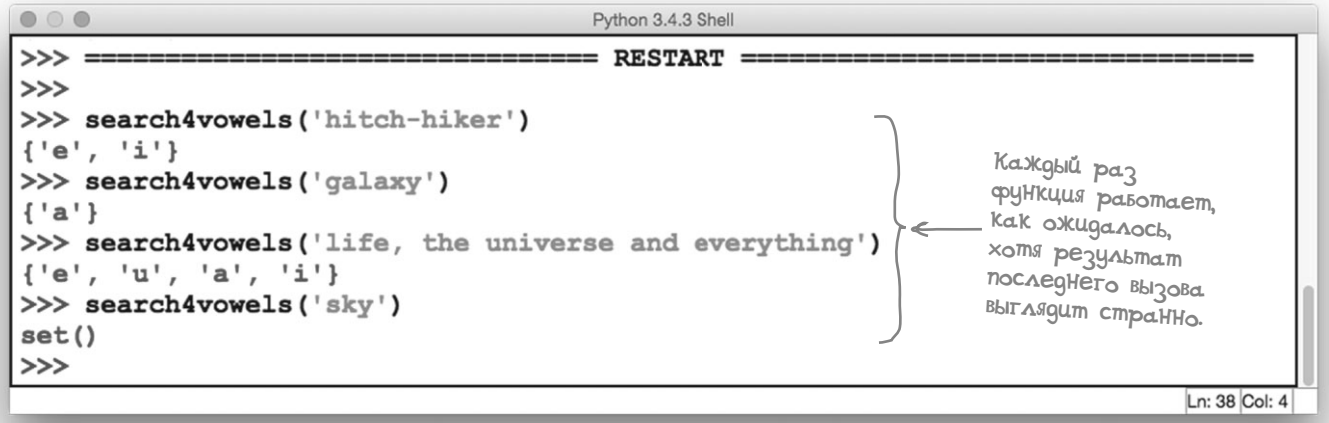
Возврат данных без использования переменной «found».

Теперь функция возвращает множество гласных, найденных в слове, что и требовалось сделать.

Однако, когда мы ее тестировали, один из результатов нас удивил...

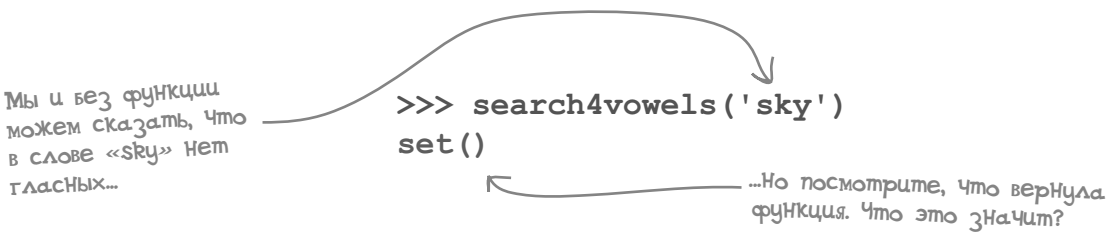
Пробная поездка

Возьмем последнюю версию функции `search4vowels` и посмотрим, как она себя ведет. Нажмите в окне редактора IDLE клавишу F5, чтобы импортировать функцию в консоль Python, а затем вызовите эту функцию несколько раз.



Что такое «set()»?

Каждый пример из нашей «Пробной поездки» работает отлично, функция принимает единственный строковый аргумент и возвращает множество найденных гласных. Результат всегда один: множество, содержащее несколько значений. Однако последний пример кажется странным, не так ли? Посмотрим на него внимательнее.



Может, вы думали, что функция должна вернуть `{}` для представления пустого множества, однако это не так, потому что `{}` — это пустой словарь, а *не* пустое множество.

Пустое множество выглядит как `set()`.

Может, это немного странно, но именно так и происходит в Python.

Давайте вспомним четыре встроенных структуры данных и то, как каждая из них, будучи пустой, представляется интерпретатором.

Вспомним встроенные структуры данных

Напомним себе все четыре встроенные структуры данных, доступные нам. Возьмем каждую из этих структур: список, словарь, множество и, наконец, кортеж.

Создадим в консоли пустую структуру каждого типа с помощью встроенных функций, затем запишем немного данных в каждую из них и выведем на экран их содержимое после этих манипуляций.

```
>>> l = list()
>>> l
[]
>>> l = [ 1, 2, 3 ]
>>> l
[1, 2, 3]
```

Используем встроенную функцию «list» для создания пустого списка, а затем присвоим данные.

Пустой список.

```
>>> d = dict()
>>> d
{}
>>> d = { 'first': 1, 'second': 2, 'third': 3 }
>>> d
{'second': 2, 'third': 3, 'first': 1}
```

Используем встроенную функцию «dict» для определения пустого словаря, а затем присвоим данные.

Пустой словарь.

```
>>> s = set()
>>> s
set()
>>> s = {1, 2, 3}
>>> s
{1, 2, 3}
```

Используем встроенную функцию «set» для определения пустого множества, а затем присвоим данные.

Пустое множество.

Множества заключаются в фигурные скобки, так же как словари. Однако пара фигурных скобок уже используется для представления пустого словаря, поэтому пустое множество представлено как «*set()*».

```
>>> t = tuple()
>>> t
()
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
```

Используем встроенную функцию «tuple» для создания пустого кортежа, а затем присвоим данные.

Пустой кортеж

Задержитесь еще немного на этой странице и посмотрите, как интерпретатор представляет каждую из пустых структур данных.

Используйте аннотации для документирования кода

Наш обзор структур данных подтвердил, что функция `search4vowels` возвращает множество. Но как пользователи смогут узнать об этом заранее, не вызывая функцию и не проверяя тип результата? Как они узнают, чего им ожидать?

Решение состоит в том, чтобы добавить эту информацию в строку документации. Вы должны четко указать в строке документации типы аргументов и возвращаемого значения, чтобы эту информацию было легко найти. Добиться согласия программистов по вопросу о том, как должна выглядеть подобная документация, очень проблематично (PEP 257 только предлагает *формат* строк документации), поэтому теперь Python 3 поддерживает так называемые **аннотации** (известные как *подсказки типа*). Аннотации документируют — стандартным способом — тип возвращаемого значения, а также типы всех аргументов. Пожалуйста, помните следующее.

1. **Аннотации функций не являются обязательными.**
   Их можно не использовать. Фактически большинство существующего кода на Python обходится без них (потому что они появились в Python 3 относительно недавно).
2. **Аннотации функций носят информационный характер.**
   Они описывают особенности функции, но не влияют на ее поведение (проверка типов не производится).

Давайте добавим аннотацию для аргументов функции `search4vowels`. Первая аннотация указывает, что в аргументе `word` функция должна получать строку (`: str`), а вторая указывает, что функция возвращает множество (`->set`):

```
def search4vowels(word:str) -> set:
    """Возвращает гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

Синтаксис аннотаций прост. К каждому аргументу функции дописывается двоеточие, за которым следует ожидаемый тип. В нашем примере: `str` означает, что функция ожидает получить строку в аргументе. Тип возвращаемого значения указывается после списка аргументов, он обозначается стрелкой, за которой следуют тип возвращаемого значения и двоеточие. Здесь `-> set:` указывает, что функция возвращает множество.

Пока все хорошо.

Мы добавили в функцию стандартные аннотации. С их помощью программисты, которые будут использовать эту функцию, узнают, чего от нее ожидать и чего функция ждет от них. Однако интерпретатор **не будет** проверять, что параметр функции — строка или что функция всегда возвращает множество. Это приводит к довольно очевидному вопросу...

Более подробно аннотации описываются в PEP 3107: https://www.python.org/dev/peps/pep-3107/.

Зачем использовать аннотации функций?

Если интерпретатор Python не использует аннотации для проверки типов аргументов и возвращаемых значений функций, зачем вообще связываться с аннотациями?

Цель аннотаций — вовсе *не* упростить жизнь интерпретатору; аннотации существуют, чтобы упростить жизнь пользователям функции. Аннотации — это **стандарт документирования**, а *не* механизм проверки типов.

Фактически интерпретатор не волнует, какие аргументы у вас есть и каков тип возвращаемого значения функции. Интерпретатор вызывает функцию с любыми переданными ей аргументами (не обращая внимания на их тип), выполняет код функции, а затем возвращает вызывающему коду любое значение, указанное в инструкции `return`. Тип возвращаемых данных не анализируется интерпретатором.

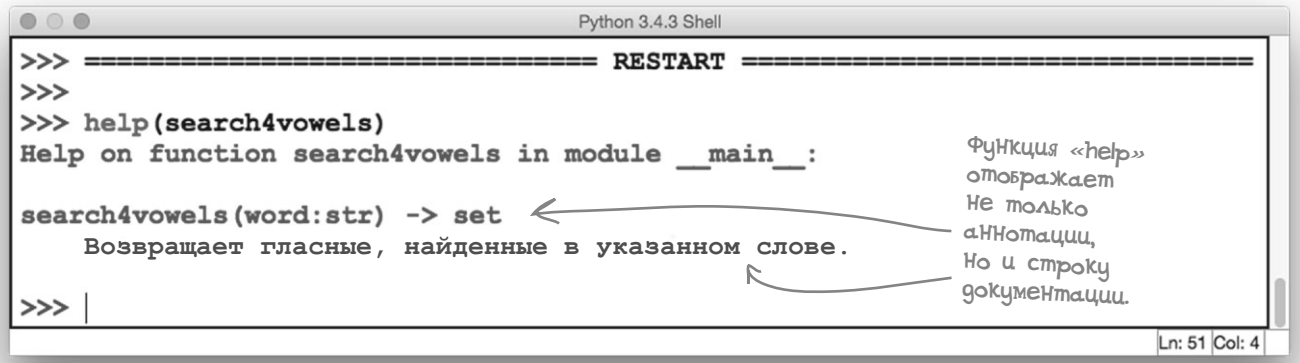
Аннотации освобождают программистов, пользующихся вашим кодом, от необходимости читать весь код функции, чтобы узнать, какие аргументы она принимает и какой объект возвращает, — вот какую пользу они приносят. Любая, даже распрекрасно написанная строка документации не избавит других программистов от чтения вашего кода, а аннотации избавят.

Все это приводит к другому вопросу: как увидеть аннотации, не читая код функции? В редакторе IDLE нажмите F5, а затем используйте встроенную функцию `help` в консоли.

Используйте аннотации для документирования ваших функций, используйте встроенную функцию «help» для просмотра аннотаций.

Пробная поездка

Если вы еще этого не сделали, добавьте аннотации в функцию `search4vowels` в окне редактора IDLE, сохраните свой код, затем нажмите F5. Консоль Python перезагрузится и выведет приглашение `>>>`, ожидая ваших приказаний. Вызовите встроенную функцию `help`, чтобы увидеть документацию для `search4vowels`:



Функции: что мы уже знаем

Давайте прервемся и вспомним, что мы уже знаем о функциях Python.

КОНТРОЛЬНЫЙ СПИСОК

- Функции — именованные фрагменты кода.
- Ключевое слово `def` определяет имя функции, а код функции расположен ниже и выделен отступами.
- В Python строки в тройных кавычках могут использоваться в роли многострочных комментариев, описывающих назначение функций. Такие комментарии называются *строками документации*.
- Функции могут принимать любое количество аргументов или не принимать их вовсе.
- Инструкция `return` позволяет функциям возвращать значения.
- Аннотации функций могут использоваться для документирования типов аргументов и возвращаемого значения функции.

Остановимся на минуту и еще раз посмотрим на код функции `search4vowels`. Теперь, когда она принимает один аргумент и возвращает множество, она стала более полезной, чем в первых версиях. Мы можем использовать ее в разных случаях.

Последняя версия функции.

```
def search4vowels(word:str) -> set:
    """Возвращает гласные, найденные в указанном слове."""
    vowels = set('aeiou')
    return vowels.intersection(set(word))
```

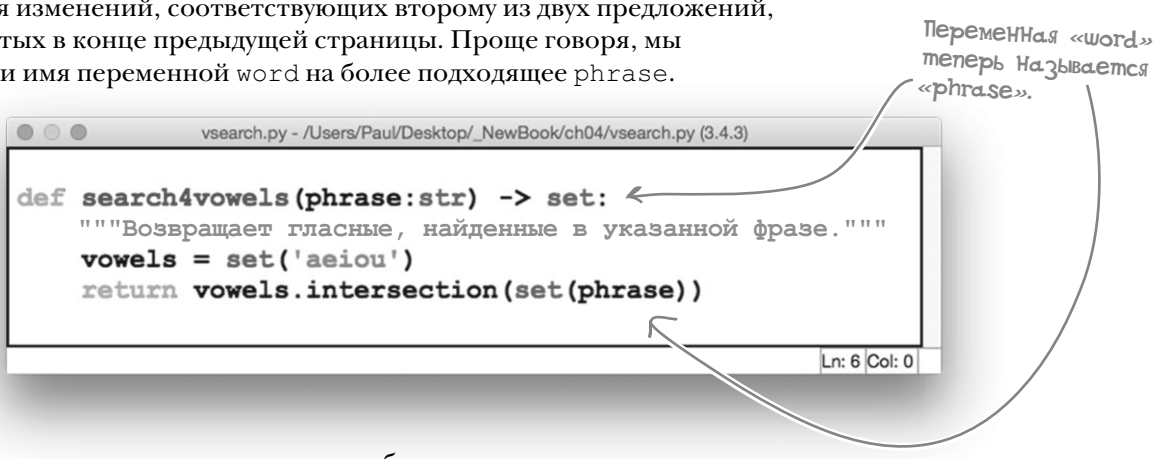
Эта функция станет полезнее, если будет принимать не только слово анализа, но и аргумент, описывающий, что именно нужно искать. Тогда мы сможем искать различные множества букв, а не только пять гласных.

Кроме того, использование имени `word` для аргумента — хорошо, но не здорово, потому что функция принимает *любую* строку в качестве аргумента, а не только одно слово. Имя `phrase` точнее соответствует тому, что ожидается получить от пользователя.

Давайте изменим нашу функцию, чтобы она соответствовала нашему новому решению.

Создание универсальной и полезной функции

Вот версия функции `search4vowels` (в редакторе IDLE) после внесения изменений, соответствующих второму из двух предложений, выдвинутых в конце предыдущей страницы. Проще говоря, мы изменили имя переменной `word` на более подходящее `phrase`.



Другое предложение — сделать так, чтобы пользователи могли определять множество букв для поиска. Тогда функция будет искать не только пять гласных. Для этого добавим в функцию второй аргумент, содержащий буквы для поиска в `phrase`. Это изменение очень легко внести. Однако, как только мы это сделаем, имя функции станет неверным, потому что теперь она будет искать не гласные, а просто буквы. Поэтому мы не будем изменять эту функцию, а создадим новую. Вот что мы предлагаем.

1. **Дать новой функции более обобщенное имя.**
   Вместо того чтобы продолжать изменять `search4vowels`, создадим новую функцию с именем `search4letters`, поскольку это название лучше отражает новое назначение функции.

2. **Добавить второй аргумент.**
   Второй аргумент позволит передать множество букв для поиска в строке. Назовем его `letters`. И не забудьте добавить аннотацию для `letters`.

3. **Удалить переменную `vowels`.**
   Переменная `vowels` в теле функции больше не нужна, потому что теперь будет использоваться определяемое пользователем множество `letters`.

4. **Обновить строку документации.**
   Когда код изменяется, нужно изменить и документацию к нему. Документация должна отражать то, что делает код в данный момент.

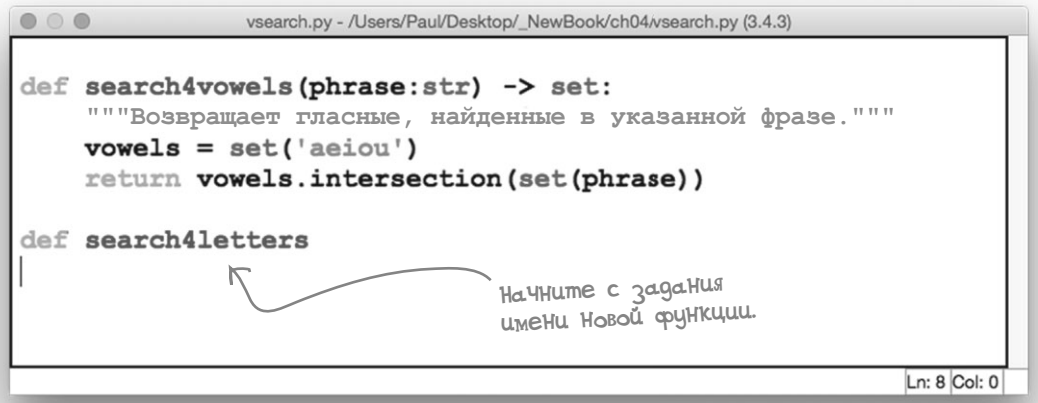
Мы будем работать над решением сразу четырех задач. По мере обсуждения очередной задачи вносите соответствующие изменения в свой файл `vsearch.py`.

Создание другой функции, 1 из 3

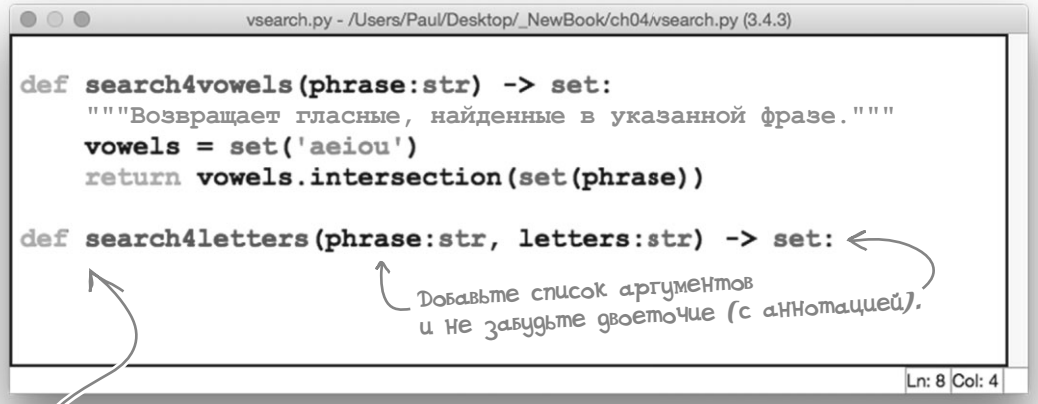
Если вы еще этого не сделали, откройте файл `vsearch.py` в окне редактора IDLE.

Шаг 1 включает создание новой функции с именем `search4letters`. PEP 8 предлагает между функциями верхнего уровня оставлять две пустые строки. Все примеры к этой книге, доступные для загрузки, соответствуют этой рекомендации, однако на печатных страницах — нет (поскольку нам нужно экономить место).

После первой функции введите `def` и имя новой функции:



В **шаге 2** завершим строку `def`, добавив имена двух аргументов функции: `phrase` и `letters`. Не забудьте заключить список параметров в круглые скобки, а также добавить двоеточие в конце (и аннотацию):



Обратили внимание, как редактор IDLE участвует в создании необходимых отступов (и автоматически позиционирует курсор)?

После завершения шагов 1 и 2 мы готовы написать код функции. Он будет похож на код в функции `search4vowels`, за исключением переменной `vowels`, которую мы собираемся удалить.

Создание другой функции, 2 из 3

В **Шаг 3** мы пишем код функции и исключаем необходимость использования переменной `vowels`. Мы можем продолжать использовать эту переменную, но ей нужно дать новое имя (потому что имя `vowels` больше не соответствует тому, что хранит переменная). Временная переменная здесь больше не нужна по той же самой причине, по которой нам больше не нужна переменная `found`. Взгляните на новую строку кода в `search4letters`, которая выполняет ту же работу, что и две строки в `search4vowels`:

Две строки кода превращаются в одну.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Возвращает гласные, найденные в указанной фразе."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    return set(letters).intersection(set(phrase))
```

Ln: 9 Col: 0

Если эта единственная строка кода `search4letters` смущает вас, не отчаивайтесь. Она проще, чем кажется. Давайте пройдемся по этой строке и посмотрим, как она работает. Сначала значение аргумента `letters` превращается в множество:

```
set(letters)
```

Создается объект множества из «letters».

Этот вызов встроенной функции `set` создает объект множества из символов в переменной `letters`. Нам не нужно присваивать этот объект переменной, потому что мы больше заинтересованы в использовании множества букв сразу, «на лету». Чтобы использовать только что созданный объект множества, поставьте после него точку, а потом укажите метод, который вы хотите использовать, потому что даже объекты, которые не присвоены переменным, имеют методы. Как мы знаем из прошлой главы, метод `intersection` принимает множество символов, содержащихся в аргументе (`phrase`), и ищет пересечение его с существующим объектом множества (`letters`) .

Найдем пересечение множества, созданного из «letters», с множеством, созданным из «phrase».

```
set(letters).intersection(set(phrase))
```

И наконец, результат пересечения возвращается вызывающему коду с помощью инструкции `return`.

Возвращаем результат вызывающему коду.

```
return set(letters).intersection(set(phrase))
```

Создание другой функции, 3 из 3

Осталось сделать **шаг 4** и добавить в новую функцию строку документации. Для этого вставьте строку в тройных кавычках после строки с `def`. Вот что у нас вышло (как и положено комментариям, содержимое строки получилось кратким и емким):

Строка документации.

```
vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

def search4vowels(phrase:str) -> set:
    """Возвращает гласные, найденные в указанной фразе."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str) -> set:
    """Возвращает множество букв из 'letters', найденных
     в указанной фразе."""
    return set(letters).intersection(set(phrase))

Ln: 10 Col: 0
```

Теперь все четыре шага пройдены и функция `search4letters` готова к тестированию.

И зачем это? Столько проблем — и все ради создания однострочной функции? Разве не проще сразу вставить строку туда, где она нужна?

Функции, кроме всего прочего, могут скрывать сложность.

Подмечено *верно*. Мы создали всего лишь однострочную функцию, которая не позволяет ощутимо сократить объем кода. Однако обратите внимание: функция содержит довольно сложную строку кода, который скрыт от пользователя функции. Такой подход является хорошей практикой программирования (не говоря уже о том, что это гораздо лучше, чем копирование и вставка).

Например, большинство программистов смогут догадаться, что делает функция `search4letters`, если встретят ее вызов где-нибудь в программе. Однако если они увидят такую сложную строку кода в программе, им придется потрудиться, чтобы определить, что она делает. Поэтому хотя функция `search4letters` короткая, все равно ее правильнее вынести в отдельную функцию, чтобы создать завершенную абстракцию.

Пробная поездка

Сохраните снова файл `vsearch.py`, затем нажмите F5, чтобы загрузить функцию `search4letters` в оболочку:

```
Python 3.4.3 Shell
>>> ========== RESTART ==========
>>>
>>> help(search4letters)
Help on function search4letters in module __main__:

search4letters(phrase:str, letters:str) -> set
    Возвращает множество букв из 'letters', найденных
    в указанной фразе.

>>> search4letters('hitch-hiker', 'aeiou')
{'e', 'i'}
>>> search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> search4letters('life, the universe, and everything', 'o')
set()
>>>
                                                    Ln: 78 Col: 4
```

Используйте встроенную функцию «help», чтобы получить справочную информацию о «search4letters».

Все эти примеры работают так, как мы и ожидали.

Функция `search4letters` получилась более универсальной, чем `search4vowels`, потому что принимает *любое* множество букв и ищет их в указанной фразе, а не только буквы `a, e, i, o, u`. Поэтому новая функция более полезная, чем `search4vowels`. Теперь представим, что у нас есть большая кодовая база, в которой `search4vowels` используется повсеместно. Было принято решение отказаться от `search4vowels` и заменить ее функцией на `search4letters`, потому что нет смысла поддерживать обе функции, если `search4letters` может то же, что и `search4vowels`. Простой поиск с заменой не даст желаемого результата, потому что с каждой заменой необходимо также добавить второй аргумент, который всегда будет строкой `'aeiou'`, чтобы поведение `search4letters` полностью соответствовало поведению `search4vowels`. Вот, например, вызов функции с одним аргументом.

```
search4vowels("Don't panic!")
```

Теперь нам нужно заменить его вызовом с двумя аргументами (автоматизированно это сделать нелегко).

```
search4letters("Don't panic!", 'aeiou')
```

Было бы неплохо каким-то способом определить *значение по умолчанию* для второго аргумента `search4letters`, которое использовалось бы в отсутствие альтернативного значения. Если мы сможем определить значение по умолчанию `'aeiou'`, то легко сможем применить операцию поиска с заменой.

Задание значений по умолчанию для аргументов

Любой аргумент в функции Python может иметь значение по умолчанию, которое будет использоваться автоматически, если при вызове функции не указать альтернативное значение. Механизм присваивания значения по умолчанию аргументу очень прост: нужно просто присвоить аргументу значение по умолчанию в строке с `def`.

Вот так сейчас выглядит строка `def` в объявлении функции `search4letters`:

```
def search4letters(phrase:str, letters:str) -> set:
```

Эта версия функции ожидает *ровно* два аргумента, первый — `phrase`, второй — `letters`. Однако если присвоить значение по умолчанию аргументу `letters`, строка с `def` для этой функции будет выглядеть так:

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Аргументу «letters» присвоено значение по умолчанию, оно будет использоваться везде, где вызывающий код не передает альтернативное значение.

Мы можем продолжить использовать функцию `search4letters` так же, как раньше: подставив при необходимости оба параметра в вызов. Но если мы забудем указать второй параметр (`letters`), интерпретатор сам подставит значение `'aeiou'`.

Внеся эти изменения в код `vsearch.py` (и сохранив его), мы сможем вызывать наши функции следующим образом:

Эти три вызова функции дают один и тот же результат.

```
>>> search4letters('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
>>> search4letters('life, the universe, and everything', 'aeiou')
{'a', 'e', 'i', 'u'}
>>> search4vowels('life, the universe, and everything')
{'a', 'e', 'i', 'u'}
```

В этом вызове мы используем «search4vowels», а не «search4letters».

Эти вызовы функций не только производят один и тот же результат, они также демонстрируют, что `search4vowels` больше не нужна, потому что аргумент `letters` в функции `search4letters` поддерживает значение по умолчанию (сравните первый и последний вызовы).

Если теперь нам понадобится отказаться от `search4vowels` и заменить все ее вызовы в нашей огромной кодовой базе вызовами `search4letters`, наше понимание механизма значений по умолчанию для аргументов функций позволит добиться желаемого простым поиском с заменой. Нам не придется использовать `search4letters` только для поиска гласных. Второй аргумент позволяет передать *любое* множество букв для поиска. Как следствие, теперь функция `search4letters` стала более гибкой и более полезной.

Позиционные и именованные аргументы

Мы уже видели, что функцию `search4letters` можно вызвать с одним или двумя аргументами, второй аргумент является необязательным. Если указать только один аргумент, параметру `letters` будет присвоена строка с гласными. Взгляните еще раз на строку определения функции:

Определение нашей функции.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

Кроме значений по умолчанию, интерпретатор Python поддерживает **именованные аргументы**. Чтобы понять, что такое именованный аргумент, рассмотрим, как происходит вызов `search4letters`:

```
search4letters('galaxy', 'xyz')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

В этом вызове две строки присваиваются параметрам `phrase` и `letters` согласно их позициям. То есть первая строка будет присвоена первому параметру (`phrase`), а вторая — второму (`letters`). Это — **позиционное присваивание**. Оно основано на порядке следования аргументов.

В Python можно также ссылаться на параметры по их именам, в таком случае порядок следования аргументов уже не важен. Это называется **присваиванием по ключу**. Чтобы им воспользоваться, при вызове функции присвойте каждую строку *в любом порядке* правильным параметрам по их именами, как показано далее.

Порядок аргументов не важен, если присваивать их по именам параметров.

```
search4letters(letters='xyz', phrase='galaxy')

def search4letters(phrase:str, letters:str='aeiou') -> set:
```

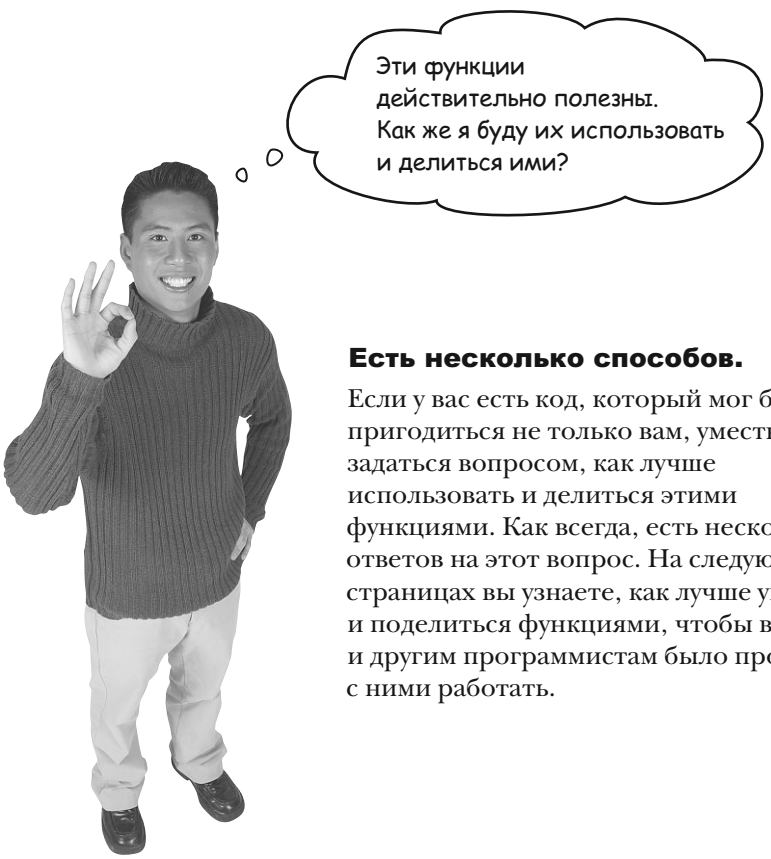
Оба вызова функции `search4letters` на этой странице дают один и тот же результат: множество, содержащее буквы `y` и `z`. Сложно оценить преимущества использования именованных аргументов на примере нашей маленькой функции `search4letters`, однако гибкость этого подхода становится очевидной, когда дело доходит до вызовов функций с большим количеством аргументов. В конце главы мы рассмотрим пример такой функции (из стандартной библиотеки).

Повторим, что мы узнали о функциях

Теперь, после ознакомления с некоторыми особенностями аргументов, повторим, что нам известно о функциях.

КОНТРОЛЬНЫЙ СПИСОК

- Кроме повторного использования кода, функции позволяют скрывать сложности. Если у вас есть сложная строка кода, которую вы собираетесь использовать, спрячьте ее за вызовом простой функции.
- Любому аргументу функции можно назначить значение по умолчанию в строке определения `def`. В этом случае при вызове функции можно не указывать аргумент.
- Кроме присваивания аргументов по позициям, можно использовать присваивание по ключам. В этом случае порядок следования аргументов не имеет значения (поскольку присваивание по именам параметров устраняет всякую неоднозначность и делает порядок следования аргументов неважным).



Есть несколько способов.

Если у вас есть код, который мог бы пригодиться не только вам, уместно задаться вопросом, как лучше использовать и делиться этими функциями. Как всегда, есть несколько ответов на этот вопрос. На следующих страницах вы узнаете, как лучше упаковать и поделиться функциями, чтобы вам и другим программистам было проще с ними работать.

Из функций рождаются модули

модуль

Потрудившись над созданием функции, которую можно использовать многократно (или двух функций — ведь в файле `vsearch.py` их две), мы озадачимся закономерным вопросом: *как лучше всего поделиться функциями?*

Функциями можно поделиться, просто скопировав блок кода из одного проекта в другой, но это плохая идея, и мы не будем ее обсуждать. Создание нескольких копий одной и той же функции загрязняет код, и это обязательно приведет к бедствию (стоит только захотеть изменить алгоритм ее работы). Гораздо лучше создать **модуль**, включающий единственную, каноническую копию всех функций, которыми вы желаете поделиться. И тут возникает еще один вопрос: *как в Python создаются модули?*

Делитесь функциями в модулях.

Ответ на удивление прост: модуль — это любой файл, содержащий функции. Кстати, это означает, что `vsearch.py` — *уже* модуль. Взгляните на него еще раз, вот он во всей красе.

vsearch.py - /Users/Paul/Desktop/_NewBook/ch04/vsearch.py (3.4.3)

```
def search4vowels(phrase:str) -> set:
    """Возвращает гласные, найденные в указанной фразе."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Возвращает множество букв из 'letters', найденных
     в указанной фразе."""
    return set(letters).intersection(set(phrase))
```

Ln: 10 Col: 0

В файле «vsearch.py» содержатся функции, что делает этот файл полноправным модулем.

Создание модулей: проще не бывает, но...

Создать модуль проще простого. Создайте файл и добавьте в него любые функции, которыми желаете поделиться.

Сделать существующий модуль доступным для программы тоже несложно. Достаточно импортировать его с использованием инструкции `import`.

Все это само по себе несложно. Однако интерпретатор предполагает, что модуль будет находиться в **пути поиска**, а гарантировать это может быть непросто. Давайте рассмотрим подробнее, как выполняется импортирование модулей.

Как происходит поиск модулей?

модуль

Вспомните, как в первой главе мы импортировали и использовали функцию `randint` из модуля `random`, который является частью стандартной библиотеки Python. Вот что мы делали в командной строке:

Импортировать модуль, а затем...

```
>>> import random
>>> random.randint(0, 255)
42
```

...вызвать функцию из этого модуля.

Что происходит во время импорта, достаточно детально описано в документации по Python, к которой вы можете обратиться за подробностями. Однако пока вам достаточно знать, что интерпретатор ищет модули в трех местах.

1. **Текущий рабочий каталог.**
   Это папка, которую интерпретатор считает текущей.

2. **Каталоги хранилища сторонних пакетов.**
   Это каталоги, в которых хранится большинство сторонних модулей Python, которые вы установили (или написали сами).

3. **Каталоги стандартной библиотеки.**
   Это каталоги, содержащие все модули, составляющие стандартную библиотеку.

Для умников

В зависимости от типа операционной системы, местоположение файлов может называться **каталогом** или **папкой**. Мы будем использовать термин «папка», если только мы не обсуждаем *текущий рабочий каталог* (уже привычный термин).

Порядок, в котором интерпретатор просматривает папки в пунктах 2 и 3, может зависеть от многих факторов. Но не беспокойтесь, в действительности не так важно знать, как работает механизм поиска. Важно понимать, что интерпретатор всегда *сначала* просматривает текущий каталог, и именно это может приводить к проблемам при работе над собственными модулями.

Покажем возможные проблемы на специально созданном маленьком примере. Вот что нужно сделать, прежде чем начать.

- [ ] Создать папку с именем `mymodules` для хранения модулей. Не важно, в какой части файловой системы вы создадите ее; просто убедитесь, что она существует и у вас есть права на чтение и запись в ней.

- [ ] Переместите файл `vsearch.py` в папку `mymodules`. Этот файл должен быть единственной копией файла `vsearch.py` на вашем компьютере.

Запуск Python из командной строки

модуль

Мы собираемся запустить интерпретатор Python в командной строке операционной системы (или в терминале), чтобы продемонстрировать возможные проблемы (хотя проблема может возникнуть и в IDLE).

Если вы используете одну из версий *Windows*, откройте командную строку и пользуйтесь ею, выполняя пример. Другие платформы, отличные от *Windows*, мы обсудим на следующей странице (но все равно читайте этот пример). Вызвать интерпретатор Python (за пределами IDLE) можно, набрав в командной строке *Windows* `C:\>` команду ***`py -3`***. Обратите внимание на скриншоте ниже, что перед тем как вызвать интерпретатор, мы сделали папку `mymodules` текущим рабочим каталогом, выполнив команду `cd`. Завершить работу интерпретатора можно в любой момент, набрав `quit()` в приглашении к вводу >>>:

Переход в папку «mymodules».

Запуск Python 3.

Импорт модуля.

Использование функции из модуля.

Завершение работы интерпретатора, возврат в командную строку ОС.

```
File Edit Window Help Redmond #1
C:\Users\Head First> cd mymodules

C:\Users\Head First\mymodules> py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'y', 'x'}
>>> quit()

C:\Users\Head First\mymodules>
```

Пока ничего необычного: мы успешно импортировали модуль `vsearch`, затем вызвали обе его функции, указав имя модуля перед их именами, через точку. Обратите внимание: интерактивная оболочка >>> действует в командной строке точно так же, как в IDLE (только нет подсветки синтаксиса). По сути это один и тот же интерпретатор.

Сеанс работы с интерпретатором удался, но только потому, что мы перешли в папку с файлом `vsearch.py`. В результате этого действия папка стала текущим рабочим каталогом. Мы знаем, что согласно правилам поиска модулей интерпретатор сначала просматривает текущий рабочий каталог, поэтому нет ничего удивительного, что он нашел наш модуль.

Но что случится, если модуль не будет находиться в текущем рабочем каталоге?

Неудачная попытка найти модуль приводит к ошибке ImportError

модуль

Повторите упражнение еще раз, но перед этим выйдите из папки с модулем. Посмотрим, что получится, если попытаться теперь импортировать модуль. Вот еще один пример взаимодействия с командной строкой *Windows*.

Сменим папку (в данном случае мы перешли в корневую папку).

Снова запускаем Python 3.

Пытаемся импортировать модуль...

...но на этот раз получаем ошибку!

```
File Edit Window Help Redmond #2
C:\Users\Head First> cd \

C:\>py -3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC
v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

C:\>
```

Файл `vsearch.py` уже не находится в текущем рабочем каталоге интерпретатора, потому что мы работаем в папке `mymodules`. Это означает, что файл модуля не может быть найден и импортирован, поэтому интерпретатор сообщает об ошибке `ImportError`.

Выполнив это упражнение на платформе, отличной от *Windows*, мы получим тот же результат (и в *Linux*, и в *Unix*, и в *Mac OS X*). Вот пример попытки импортировать модуль из каталога `mymodules` в OS X.

Перейдем в другую папку и наберем «python3», чтобы запустить интерпретатор.

Импортируем модуль.

Работает: мы можем использовать функцию модуля.

```
File Edit Window Help Cupertino #1
$ cd mymodules

mymodules$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
>>> vsearch.search4vowels('hitch-hiker')
{'i', 'e'}
>>> vsearch.search4letters('galaxy', 'xyz')
{'x', 'y'}
>>> quit()

mymodules$
```

Завершение работы интерпретатора Python и возврат в командную строку операционной системы.

Ошибка ImportError возникает на любой платформе

Если вы думаете, что работа на платформе, отличной от *Windows*, поможет избежать проблемы с импортом, которую мы только что наблюдали, подумайте еще раз: после смены папки такая же ошибка `ImportError` наблюдается и в UNIX-подобных системах.

модуль

Сменим папку (в данном случае мы перешли в домашний каталог).

Снова запускаем Python 3.

Пытаемся импортировать модуль...

...Но на этот раз получаем ошибку!

```
File Edit Window Help Cupertino #2
mymodules$ cd

$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import vsearch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'vsearch'
>>> quit()

$
```

Как и в случае с *Windows*, файл `vsearch.py` уже не находится в текущем рабочем каталоге интерпретатора, потому что мы работаем в папке, отличной от `mymodules`. Это означает, что файл не может быть найден и импортирован, поэтому интерпретатор сообщает об ошибке `ImportError`. Эта проблема не зависит от платформы, на которой работает Python.

Это не Глупые вопросы

В: Разве нельзя указать конкретное местоположение файла, например `import C:\mymodules\vsearch` в Windows или `import /mymodules/vsearch` в UNIX-подобных системах?

О: Нет. Конечно, очень заманчиво сделать что-то подобное, но прием не сработает, потому что нельзя использовать подобные пути в инструкции `import`. Добавлять в программы статические пути — последнее дело, потому что пути часто меняются (по целому ряду причин). Лучше избегать подобной практики, если возможно.

В: Если я не могу использовать пути, как подсказать интерпретатору, где искать модули?

О: Если интерпретатор не находит модуль в текущем рабочем каталоге, он просматривает папки со сторонними пакетами и стандартной библиотекой (подробнее о сторонних пакетах на следующей странице). Если вы можете добавить свой модуль в одну из папок со сторонними пакетами, интерпретатор сможет найти его там (не важно, по какому пути).

Добавление модуля в хранилище сторонних пакетов

модуль

Вспомним, что мы говорили о **хранилище сторонних пакетов** ранее, когда мы рассказывали о трех местах, которые при импорте просматривает интерпретатор.

2. **Каталоги хранилища сторонних пакетов**
   В этих каталогах хранится большинство сторонних модулей Python, которые вы установили (или написали сами).

Так как поддержка сторонних модулей является основой стратегии повторного использования кода в Python, неудивительно, что интерпретатор имеет встроенную возможность добавления модулей в свое окружение.

Обратите внимание: модули, входящие в состав стандартной библиотеки, поддерживаются основными разработчиками Python. Эта огромная коллекция модулей предназначена для широкого использования, но вы не должны вмешиваться в эту коллекцию. Не стоит добавлять свои модули или удалять модули из стандартной библиотеки. Однако добавление и удаление модулей в хранилище сторонних пакетов всячески приветствуется, поэтому в Python есть инструменты для упрощения этой процедуры.

Использование инструмента «setuptools» для установки пакетов

Начиная с версии Python 3.4 стандартная библиотека включает модуль `setuptools`, который используется для добавления модулей в хранилище сторонних пакетов. Процедура распространения модулей первое время может казаться сложной, но мы попробуем установить `vsearch` в хранилище сторонних пакетов с помощью `setuptools`. От нас потребуется сделать всего три шага.

1. **Создать описание дистрибутива.**
   Подобное описание идентифицирует модуль, который мы хотим установить с помощью `setuptools`.
2. **Сгенерировать файл дистрибутива.**
   Используя Python в командной строке, создадим файл, содержащий код нашего модуля.
3. **Установить файл дистрибутива.**
   Снова используя Python в командной строке, установим файл (который включает наш модуль) в хранилище сторонних пакетов.

В версии Python 3.4 (или выше) работать с setuptools намного проще. Если у вас старая версия, лучше обновитесь.

В шаге 1 требуется создать (как минимум) два файла с описанием нашего модуля: `setup.py` и `README.txt`. Посмотрим, что они содержат.

Создание необходимых файлов установки

- ☐ Создание файла описания.
- ☐ Создание файла дистрибутива.
- ☐ Установка файла дистрибутива.

Будем отмечать каждый из этих пунктов по мере выполнения.

Выполнив три шага, перечисленных на предыдущей странице, мы получим **дистрибутив** модуля. Это единственный сжатый файл, содержащий все необходимое для установки модуля в хранилище сторонних пакетов.

Чтобы выполнить шаг 1, *создание описания дистрибутива*, нам нужно создать два файла, которые мы поместим в ту папку, где находится файл `vsearch.py`. Это нужно сделать независимо от платформы, на которой вы работаете. Первый файл, который должен называться `setup.py`, описывает некоторые детали.

Ниже приводится файл `setup.py`, который мы создали для описания модуля в файле `vsearch.py`. Он содержит две строки кода на Python: первая импортирует функцию `setup` из модуля `setuptools`, вторая вызывает функцию `setup`.

Функция `setup` имеет большое число аргументов, многие из которых необязательны. Обратите внимание: для улучшения читаемости кода вызов функции `setup` разнесен на девять строк. Мы воспользовались поддержкой именованных аргументов в Python, чтобы четко видеть, какое значение присваивается какому параметру. Наиболее важные аргументы выделены; первый определяет имя дистрибутива, а второй перечисляет все файлы `.py` с исходными кодами, которые будут включены в пакет.

Импорт функции «setup» из модуля «setuptools».

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

Аргумент «name» определяет имя дистрибутива. Дистрибутивы часто называют так же, как модули.

Вызов функции «setup», мы разнесли его на несколько строк.

Список файлов «.py», которые нужно включить в пакет. В примере такой файл только один: «vsearch».

Кроме `setup.py` механизм `setuptools` требует наличия еще одного файла — `readme`. В него можно поместить текстовое описание пакета. Наличие этого файла обязательно, но его содержимое может быть любым, поэтому вы можете создать пустой файл `README.txt` в той же папке, что и `setup.py`. Этого достаточно, чтобы выполнить шаг 1.

Создание файла дистрибутива

- [x] Создание файла описания.
- [ ] Создание файла дистрибутива.
- [ ] Установка файла дистрибутива.

На этом этапе в папке `mymodules` должны быть три файла `vsearch.py`, `setup.py` и `README.txt`.

Мы готовы создать из них пакет дистрибутива. В нашем списке это шаг 2: *создание файла дистрибутива*. Мы выполним его в командной строке. Сделать это просто, но на разных платформах команды будут немного различаться, то есть в *Windows* это будут одни команды, а в UNIX-подобных системах (*Linux, Unix, Mac OS X*) — немного другие.

Создание файла дистрибутива в Windows

Если вы пользуетесь *Windows*, откройте командную строку в папке с этими тремя файлами, а затем введите команду:

Запускаем Python 3 в Windows.

```
C:\Users\Head First\mymodules> py -3 setup.py sdist
```

Выполняем код в «setup.py»...

...и передаем аргумент «sdist».

Интерпретатор Python выполнит эту работу сразу, как только вы введете команду. Вы увидите на экране довольно много сообщений (мы покажем их в сокращенной форме).

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
creating dist
creating 'dist\vsearch-1.0.zip' and adding 'vsearch-1.0' to it
adding 'vsearch-1.0\PKG-INFO'
adding 'vsearch-1.0\README.txt'
      ...
adding 'vsearch-1.0\vsearch.egg-info\top_level.txt'
removing 'vsearch-1.0' (and everything under it)
```

Если вы видите это сообщение, все прошло успешно. Если появилось сообщение об ошибке, убедитесь, что пользуетесь версией Python не ниже 3.4, а также проверьте содержимое своего файла «setup.py» — оно должно совпадать с примером.

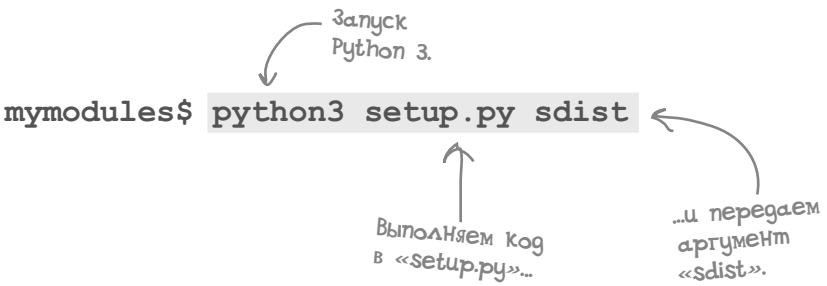
Когда снова появится приглашение командной строки *Windows*, ваши три файла уже будут упакованы в один **файл дистрибутива**. Это файл пригоден для установки, в нем находится исходный код модуля. В примере файл получил имя `vsearch-1.0.zip`.

Вы найдете новый ZIP-файл в папке `dist`, которая тоже была создана модулем `setuptools` внутри рабочей папки (в нашем случае это `mymodules`) .

Файлы дистрибутивов в UNIX-подобных ОС

- [x] Создание файла описания.
- [ ] Создание файла дистрибутива.
- [ ] Установка файла дистрибутива.

Если вы работаете не в *Windows*, то можете создать файл дистрибутива в схожей манере. Находясь в папке с тремя файлами (`setup.py`, `README.txt` и `vsearch.py`), выполните в командной строке своей операционной системы следующую команду.



Как и в *Windows*, на экране появятся сообщения:

```
running sdist
running egg_info
creating vsearch.egg-info
      ...
running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
      ...
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

Сообщение отличается от того, что видят пользователи Windows. Если вы видите такое сообщение, все прошло успешно. Если нет, проверьте все еще раз (так же как в Windows).

Когда снова появится приглашение командной строки, ваши три файла уже будут упакованы в один **файл дистрибутива** (как того требует аргумент `sdist`). Файл пригоден для установки, в нем находится исходный код модуля; в примере файл получил имя `vsearch-1.0.tar.gz`.

Вы найдете новый файл архива в папке `dist`, которая тоже была создана модулем `setuptools` внутри рабочей папки (в нашем случае `mymodules`).

После создания файла дистрибутива (ZIP или сжатого архива tar) его можно установить в хранилище сторонних пакетов.

Установка пакетов при помощи «pip»

- [x] Создание файла описания.
- [x] Создание файла дистрибутива.
- [ ] Установка файла дистрибутива.

Теперь, после создания файла дистрибутива, архива ZIP или tar (в зависимости от платформы), пора выполнить шаг 3: *установку файла дистрибутива*. Как и во многих подобных случаях, в Python имеются инструменты, упрощающие эту процедуру. В версии Python 3.4 (и выше) есть инструмент `pip`, **P**ackage **I**nstaller for **P**ython (инсталлятор пакетов для Python).

Шаг 3 в Windows

Найдите папку `dist` с только что созданным ZIP-файлом (этот файл называется `vsearch-1.0.zip`). В *Windows Explorer* нажмите клавишу Shift и, удерживая ее, щелкните правой кнопкой мыши на папке, чтобы открыть контекстное меню. Выберите пункт *Open command window here* (*Запуск командной строки здесь...*). В результате откроется окно командной строки *Windows*. Чтобы завершить шаг 3, введите в командной строке следующую инструкцию.

Запуск Python 3 с модулем pip и командой установить указанный ZIP-файл.

```
C:\Users\...\dist> py -3 -m pip install vsearch-1.0.zip
```

Если команда завершится с ошибкой из-за проблем с правами доступа, перезапустите командную строку с правами администратора *Windows* и выполните ее снова.

В случае успешного выполнения команды на экране появится следующее сообщение.

```
Processing c:\users\...\dist\vsearch-1.0.zip
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Успешно!

Шаг 3 в UNIX-подобных ОС

В *Linux*, *Unix* или *Mac OS X* откройте терминал в только что созданной папке `dict`, а затем наберите следующую команду.

Запуск Python 3 с модулем pip и командой установить указанный tar-файл.

```
.../dist$ sudo python3 -m pip install vsearch-1.0.tar.gz
```

Мы используем команду «sudo», чтобы гарантировать права, достаточные для установки.

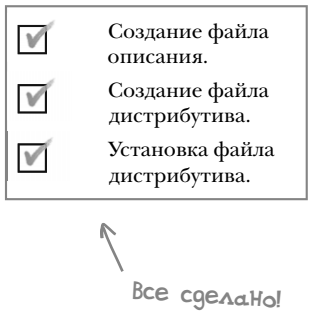
После успешного выполнения команды на экране появится следующее сообщение.

```
Processing ./vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Running setup.py install for vsearch
Successfully installed vsearch-1.0
```

Успешно!

Теперь модуль vsearch установлен как сторонний пакет.

Модули: что мы знаем



Теперь, после установки модуля `vsearch`, мы можем использовать инструкцию `import vsearch` в любых программах, потому что интерпретатор теперь сможет найти требуемый модуль.

Если позже мы решим обновить код модуля, то сможем повторить эти три шага и установить обновленный пакет в хранилище. Однако, выпуская новую версию модуля, не забудьте указать новый номер версии в файле `setup.py`.

Давайте подытожим все, что мы уже знаем о модулях.

КОНТРОЛЬНЫЙ СПИСОК

- Модуль — одна или более функций, сохраненных в файле.
- Вы можете использовать любой модуль, поместив его в *текущий рабочий каталог* (возможно, но трудновыполнимо) или в *хранилище сторонних пакетов* (гораздо лучший выбор).
- При помощи `setuptools` можно за три шага установить модуль в *хранилище сторонних пакетов*, чтобы потом импортировать модуль и вызывать его функции в своих программах независимо от того, какой каталог является *текущим рабочим* в данный момент.

Как отдать свой код (то есть поделиться)

Теперь у вас есть готовый файл дистрибутива, и вы можете поделиться этим файлом с другими программистами, позволив им установить ваш модуль с помощью `pip`. Отдать файл можно официально или неофициально.

Выбрав неофициальный путь, вы можете передавать его как угодно и кому угодно (например, по электронной почте, записав на USB-накопитель или выложив на собственном веб-сайте). Все на ваш выбор.

При помощи pip любой программист на Python установит ваш модуль.

Чтобы поделиться модулем официально, нужно выгрузить дистрибутив в веб-репозиторий Python, который управляется централизованно. Репозиторий называется PyPI (произносится «пай пи ай») — *Python Package Index* (каталог пакетов для Python). Сайт существует для того, чтобы программисты на Python могли делиться своими модулями. Детали изложены на сайте PyPI: ***https://pypi.python.org/pypi***. Процесс выгрузки и публикации файлов дистрибутивов в PyPI подробно описан в онлайн-руководстве, поддерживаемом *Python Packaging Authority* по ссылке ***https://www.pypa.io***. (Руководство на русском языке: ***http://itscreen.tk/blog/30-registraciya-paketa-na-pypi/*** — *Прим. ред.*)

Мы практически завершили начальное знакомство с функциями и модулями. Осталась только одна маленькая загадка, которая тоже требует нашего внимания (не больше пяти минут). Итак, переверните страницу.

Случай странного поведения аргументов функции

Роман и Юлия работают над этой главой, а теперь они спорят о поведении аргументов функции.

Роман считает, что аргументы передаются в функцию **по значению**, и он написал небольшую функцию `double`, чтобы это доказать. Функция Романа `double` работает с данными любого типа.

Вот код Романа.

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)
```

Юлия считает, что аргументы передаются в функцию **по ссылке**. Она тоже написала маленькую функцию `change`, которая работает со списками, чтобы доказать свою точку зрения.

Вот копия кода Юлии:

```
def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Роман и Юлия никогда раньше не спорили, до этого момента он были единодушны в вопросах программирования. Давайте поэкспериментируем в консоли и попытаемся понять, кто из них прав: Роман, утверждающий, что аргументы передаются по значению, или Юлия, которая считает, что они передаются по ссылке. Ведь оба не могут быть правы? Чтобы решить эту маленькую загадку, нужно ответить на вопрос:

Семантика вызова в Python поддерживает передачу аргументов по значению или по ссылке?

Для умников

Напомним, что под **передачей аргумента по значению** подразумевается передача в аргументе значения переменной. Если значение в теле функции изменится, это никак не отразится на значении переменной в вызывающем коде. В данном случае аргумент можно рассматривать как *копию* значения переменной. При **передаче аргумента по ссылке** (иногда это называется **передачей аргумента по адресу**) поддерживается связь между аргументом и переменной в вызывающем коде. Если аргумент изменится в теле функции, то переменная в вызывающем коде также изменится. В этом случае аргумент можно рассматривать как *псевдоним* переменной.

Демонстрация семантики вызова по значению

Чтобы разобраться, о чем спорят Роман и Юлия, скопируем их функции в отдельный модуль, который назовем `mystery.py`. Вот так выглядит этот модуль в окне редактора IDLE:

Функции очень похожи. Каждая принимает один аргумент, выводит его на экран, обрабатывает его значение, а затем снова выводит на экран.

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Одна функция удваивает переданный аргумент.

Другая функция добавляет строку в конец указанного списка.

Ln: 11 Col: 0

Как только Роман увидел модуль на экране, он сел за клавиатуру, нажал F5 и набрал следующую команду в консоли IDLE. После этого Роман откинулся на спинку кресла, скрестил руки и сказал: «Видишь? Я же говорил, что аргументы передаются по значению». Итак, посмотрим, как Роман вызывал свою функцию.

Роман вызвал функцию «double» три раза: один раз с целым значением, второй раз со строкой и в последний раз со списком.

```
>>> num = 10
>>> double(num)
Before:  10
After:   20
>>> num
10
>>> saying = 'Hello '
>>> double(saying)
Before:  Hello
After:   Hello Hello
>>> saying
'Hello '
>>> numbers = [ 42, 256, 16 ]
>>> double(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 42, 256, 16]
>>> numbers
[42, 256, 16]
```

Каждый вызов подтверждает, что значение, переданное в аргументе, было изменено внутри функции, но изменения не отразились на переменной в консоли. Похоже, аргументы передаются функциям по значению.

Демонстрация семантики вызова по ссылке

Очевидный успех Романа не остановил Юлию. Теперь клавиатура перешла в ее распоряжение. Покажем еще раз, как выглядит модуль в редакторе IDLE, включающий функцию `change` Юлии:

Модуль «mystery.py»

mystery.py - /Users/Paul/Desktop/_NewBook/ch04/mystery.py (3.5.0)

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)

def change(arg):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Функция Романа.

Функция Юлии.

Ln: 11 Col: 0

Юлия набрала несколько строк кода в консоли, затем откинулась на спинку кресла, скрестила руки и сказала Роману: «Ну, если Python поддерживает передачу только по значению, как ты объяснишь такое поведение?» Роман потерял дар речи.

Посмотрим, что же сделала Юлия.

Используя тот же список, что и Роман, Юлия вызвала функцию «change».

```
>>> numbers = [ 42, 256, 16 ]
>>> change(numbers)
Before:  [42, 256, 16]
After:   [42, 256, 16, 'More data']
>>> numbers
[42, 256, 16, 'More data']
```

Только посмотрите, что произошло! На этот раз изменение значения аргумента в функции отразилось на переменной в консоли. Похоже, что Python также поддерживает семантику передачи аргументов по ссылке.

Странное поведение.

Функция Романа ясно показала, что аргументы передаются по значению, а функция Юлии демонстрирует передачу по ссылке.

Как такое возможно? Что здесь происходит? Неужели Python поддерживает *оба* типа передачи?

Решено: случай странного поведения аргументов функций

Семантика вызова в Python поддерживает передачу аргументов в функцию по значению или по ссылке?

В данном случае правы *и* Роман, *и* Юлия. В зависимости от ситуации, семантика вызовов функций в **Python** поддерживает **оба** способа передачи аргументов: по значению *и* по ссылке.

Вспомните: переменные в **Python** совсем не такие, какими мы привыкли их видеть в других языках программирования; переменные — это **ссылки на объекты**. Значение, хранимое в переменной, — это адрес объекта в памяти. И в функцию передается адрес, а не фактическое значение. Вернее сказать, что функции в **Python** поддерживают *семантику вызова с передачей аргументов по ссылкам на объекты.*

В зависимости от типа объекта, на который ссылается переменная, семантика вызова функции может различаться. Так по значению или по ссылке передаются аргументы функциям Романа и Юлии? Интерпретатор определяет типы объектов, на которые ссылается переменная (адрес в памяти). Если переменная ссылается на **изменяемое** значение, применяется семантика передачи аргумента по ссылке. Если тип данных, на который ссылается переменная, **неизменяемый**, происходит передача аргумента по значению. Посмотрим, что это означает с точки зрения данных.

Списки, словари и множества (будучи изменяемыми) всегда передаются в функцию по ссылке — изменения, внесенные в них внутри функции, отражаются в вызывающем коде.

Строки, целые числа и кортежи (будучи неизменяемыми) всегда передаются в функцию по значению — изменения, внесенные в них внутри функции, не отражаются в вызывающем коде. Если тип данных неизменяемый, то функция не может его изменить.

Как же обстоит дело в следующем случае?

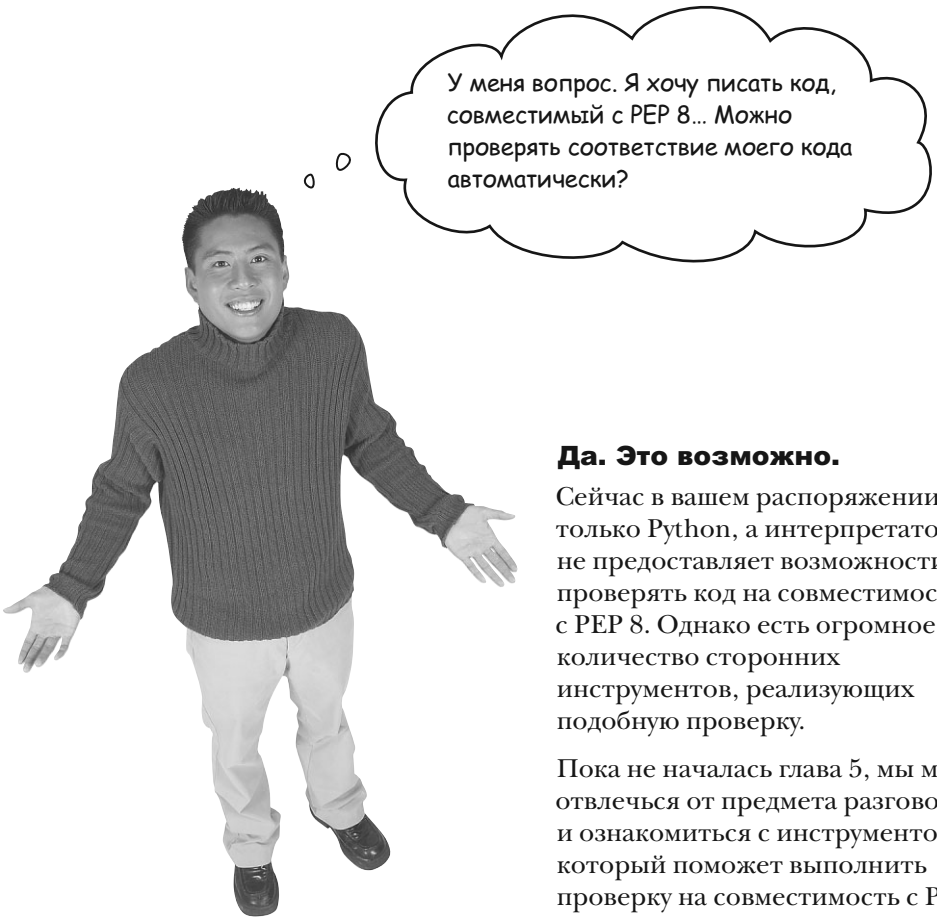
```
arg = arg * 2
```

Как получилось, что эта строка кода изменила переданный список внутри функции, а в вызывающем коде изменения оказались не видны (что привело Романа к ошибочному выводу, что все аргументы передаются по значению)? Это похоже на странную ошибку в интерпретаторе, ведь мы уже выяснили, что любые вмешательства в изменяемые значения отражаются в вызывающем коде, однако этого почему-то не произошло. Дело в том, что функция Романа *не* изменила список `numbers` в вызывающем коде, хотя списки — изменяемые типы данных. Что же произошло на самом деле?

Чтобы понять это, рассмотрим строку кода, представленную выше, которая является **инструкцией присваивания**. Вот что происходит во время присваивания: код справа от символа = выполняется *первым*, и какое бы значение ни было получено, ссылка на него присваивается переменной слева от символа =. При выполнении кода `arg * 2` создается *новое* значение, которому присваивается новая ссылка. Эта ссылка присваивается переменной `arg`, а предыдущее ее значение затирается. Однако «старая» ссылка на объект все еще существует в вызывающем коде, значение по ней не было изменено, поэтому там сохраняется прежний список, а не удвоенный, полученный в функции Романа. Функция Юлии, напротив, демонстрирует иное поведение, потому что вызывает метод `append` существующего списка. Здесь нет присваивания, и ссылка на объект не затирается. Поэтому функция Юлии изменяет список и в вызывающем коде, ведь вызывающий код и функция имеют дело со ссылкой *на один и тот же* объект.

Загадка разгадана, и мы почти готовы к главе 5. Остался еще один момент.

Как проверить соответствие рекомендациям PEP 8?



Да. Это возможно.

Сейчас в вашем распоряжении только Python, а интерпретатор не предоставляет возможности проверять код на совместимость с PEP 8. Однако есть огромное количество сторонних инструментов, реализующих подобную проверку.

Пока не началась глава 5, мы можем отвлечься от предмета разговора и ознакомиться с инструментом, который поможет выполнить проверку на совместимость с PEP 8.

Приготовьтесь к проверке на совместимость с PEP 8

Итак, отклонимся от темы и проверим, соответствует ли код рекомендациям PEP 8.

Сообщество программистов на Python потратило много времени на создание инструментов, улучшающих жизнь разработчиков. Один из таких инструментов — **pytest**, *фреймворк тестирования* программ на Python. Не важно, какие тесты вы пишете, **pytest** придет к вам на помощь. Чтобы увеличить возможности **pytest**, к нему можно добавлять плагины.

Один из таких плагинов, **pep8**, использует **pytest** для проверки соответствия кода рекомендациям PEP 8.

Узнайте больше о pytest на https://doc.pytest.org/en/latest/

Вспомним наш код

Еще раз взглянем на `vsearch.py`, прежде чем при помощи **pytest**/**pep8** проверить его совместимость с PEP 8. Кроме того, нам нужно установить оба инструмента, потому что они не поставляются вместе с Python (об этом на следующей странице).

И вот код нашего модуля `vsearch.py`, который мы собираемся проверить на совместимость с PEP 8.

```
def search4vowels(phrase:str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))

def search4letters(phrase:str, letters:str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Это код «vsearch.py».

Установка pytest и плагина pep8

Мы уже использовали инструмент `pip` для установки модуля `vsearch.py` на компьютере. Также инструмент `pip` можно использовать для установки модулей сторонних разработчиков.

Для этого нужно открыть командную строку в операционной системе (при подключении к Интернету). В главе 6 мы будем использовать `pip` для установки сторонней библиотеки, а сейчас установим с его помощью фреймворк тестирования **pytest** и плагин **pep8**.

Установка инструментов разработчика для тестирования

ОБЪЕЗД

На скриншотах ниже показаны сообщения, которые появляются в процессе установки на платформе *Windows*. В *Windows* интерпретатор Python 3 запускается командой `py -3`. В *Linux* или *Mac OS X* следует использовать команду `sudo python3`. Чтобы установить **pytest** при помощи `pip` в *Windows*, выполните `py -3` в командной строке, запущенной с правами администратора (найдите файл `cmd.exe`, щелкните на нем правой кнопки мыши и выберите в контекстном меню пункт *Run as Administrator (Запуск от имени администратора)*).

Запустить от имени администратора...

...вызвать команду «pip» для установки «pytest»...

...убедиться в успешной установке.

```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Windows\system32>py -3 -m pip install pytest
Collecting pytest
  Downloading pytest-2.8.7-py2.py3-none-any.whl (151kB)
    100% |################################| 155kB 1.3MB/s
Collecting colorama (from pytest)
  Downloading colorama-0.3.6-py2.py3-none-any.whl
Collecting py>=1.4.29 (from pytest)
  Downloading py-1.4.31-py2.py3-none-any.whl (81kB)
    100% |################################| 86kB 131kB/s
Installing collected packages: colorama, py, pytest
Successfully installed colorama-0.3.6 py-1.4.31 pytest-2.8.7

C:\Windows\system32>
```

Судя по сообщениям, сгенерированным командой `pip`, **pytest** имеет две зависимости, которые тоже были установлены (**colorama** и **py**). То же можно заметить при установке плагина **pep8**: вместе с ним устанавливаются его зависимости. Вот команда для установки плагина.

```
py  -3  -m  pip  install  pytest-pep8
```

Запомните: если вы работаете не в Windows, используйте «sudo python3» вместо «py -3».

Оставаясь в режиме администратора, выполните команду, чтобы установить плагин «pep8».

```
Administrator: Command Prompt
C:\Windows\system32>py -3 -m pip install pytest-pep8
Collecting pytest-pep8
  Downloading pytest-pep8-1.0.6.tar.gz
Collecting pytest-cache (from pytest-pep8)
  Downloading pytest-cache-1.0.tar.gz
Requirement already satisfied (use --upgrade to upgrade): pytest>=2.4.2 in c:\pr
ogram files\python 3.5\lib\site-packages (from pytest-pep8)
Collecting pep8>=1.3 (from pytest-pep8)
  Downloading pep8-1.7.0-py2.py3-none-any.whl (41kB)
    100% |################################| 45kB 174kB/s
Collecting execnet>=1.1.dev1 (from pytest-cache->pytest-pep8)
  Downloading execnet-1.4.1-py2.py3-none-any.whl (40kB)
    100% |################################| 40kB 174kB/s
Requirement already satisfied (use --upgrade to upgrade): py>=1.4.29 in c:\progr
am files\python 3.5\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Requirement already satisfied (use --upgrade to upgrade): colorama in c:\program
 files\python 3.5\lib\site-packages (from pytest>=2.4.2->pytest-pep8)
Collecting apipkg>=1.4 (from execnet>=1.1.dev1->pytest-cache->pytest-pep8)
  Downloading apipkg-1.4-py2.py3-none-any.whl
Installing collected packages: apipkg, execnet, pytest-cache, pep8, pytest-pep8
  Running setup.py install for pytest-cache ... done
  Running setup.py install for pytest-pep8 ... done
Successfully installed apipkg-1.4 execnet-1.4.1 pep8-1.7.0 pytest-cache-1.0 pyte
st-pep8-1.0.6

C:\Windows\system32>
```

Команда выполнена успешно, необходимые зависимости установлены.

Соответствует ли наш код рекомендациям в PEP 8?

ОБЪЕЗД

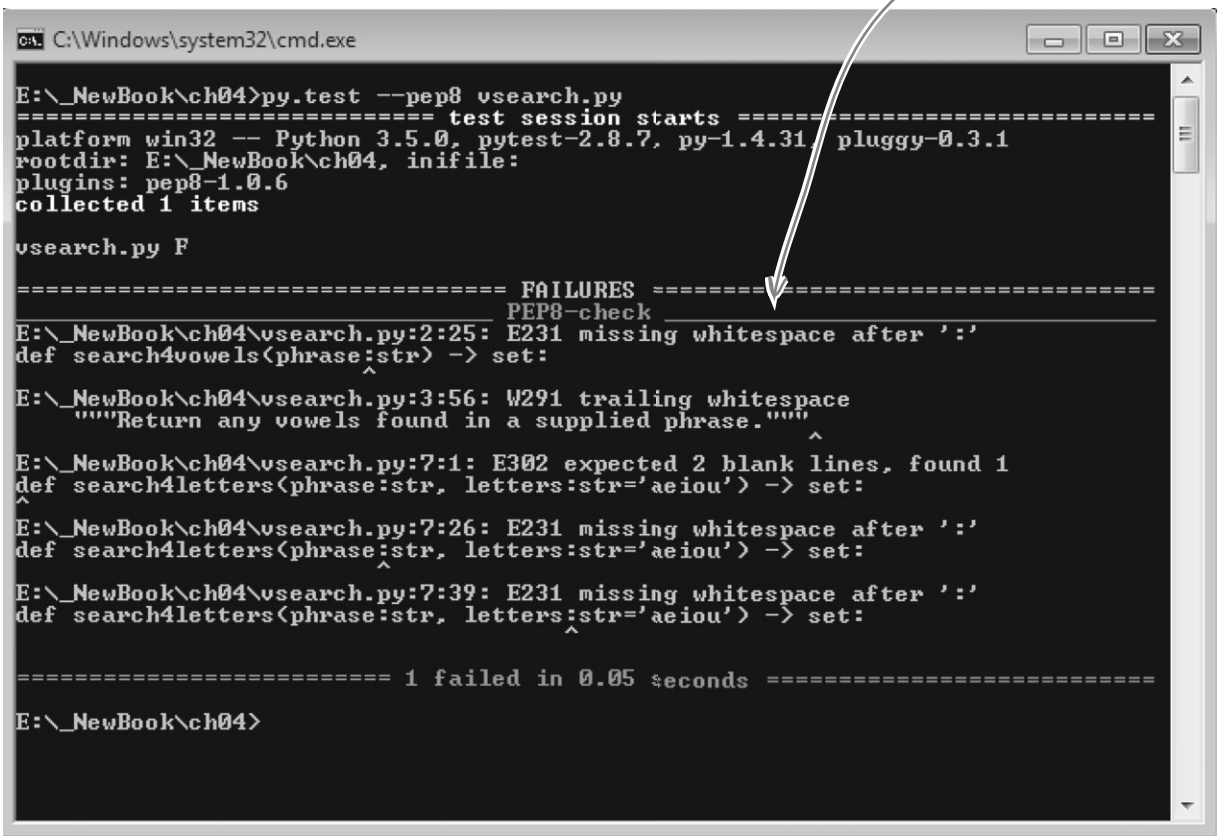
После установки **pytest** и **pep8** можно проверить код на совместимость с PEP 8. Для этого, независимо от операционной системы, нужно выполнить одну и ту же команду (от платформы зависит только процесс установки).

В процессе установки **pytest** на компьютер была установлена новая программа `py.test`. Запустим эту программу и проверим код на совместимость с PEP 8. Учтите: программа должна запускаться в том каталоге, где находится `vsearch.py`.

```
py.test  --pep8  vsearch.py
```

Вот что мы получили на компьютере с *Windows*.

Ой-ой. Если строка на экране выделена красным, это недобрый знак.



Похоже проверка прошла **неудачно**. Значит, наш код не так совместим с PEP 8, как хотелось бы.

Не пожалейте времени и прочтите сообщения, показанные здесь (или на экране, если вы выполнили пример). Оказывается, все «ошибки» относятся — по большей части — к *пробельным символам* (например, пробелам, табуляциям, символам перевода строки и т. д.). Рассмотрим их подробнее.

Разбираемся с сообщениями об ошибках

Инструменты **pytest** и **pep8** подчеркнули *пять* ошибок в коде `vsearch.py`.

На самом деле все пять проблем связаны с тем, что в трех местах мы пропустили пробел после символа `:` в аннотациях аргументов. Посмотрите на первое сообщение. Обратите внимание, что **pytest** использует символ ^, чтобы указать место проблемы.

```
            ...:2:25: E231 missing whitespace after ':'
def search4vowels(phrase:str) -> set:
                         ^
```

Описание проблемы: «...:2:25: E231 отсутствует пробельный символ после ':'».

Место обнаружения проблемы.

Следующие два сообщения в выводе **pytest** указывают, что мы допустили такую же ошибку еще в трех местах: один раз в строке 2, дважды в строке 7. Проблему легко исправить: *просто добавьте после двоеточия один пробел.*

Следующая проблема не кажется такой уж большой, но все же интерпретируется как ошибка, потому что рассматриваемая строка кода (строка 3) нарушает рекомендацию, которая требует не добавлять пробелы в конце строки.

Описание проблемы: «...:3:56: W291 завершающий пробельный символ».

```
                                ...:3:56: W291 trailing whitespace
"""Return any vowels found in a supplied phrase."""
                                                  ^
```

Место обнаружения проблемы.

Проблему в строке 3 тоже легко исправить: *удалите все пробельные символы в конце строки.*

И последняя проблема (в начале строки 7).

```
                                ...7:1: E302 expected 2 blank lines, found 1
def search4letters(phrase:str, letters:str='aeiou') -> set:
^
```

Описание проблемы: «...7:1: E302 ожидается 2 пустых строки, найдена 1».

Проблема находится в начале строки 7.

В PEP 8 есть рекомендация, касающаяся создания функций в модуле: *определения функций верхнего уровня и классов должны отделяться двумя пустыми строками.* В нашем коде функции `search4vowels` и `search4letters` являются функциями верхнего уровня в файле `vsearch.py` и отделены друг от друга единственной пустой строкой. Чтобы добиться совместимости с PEP 8, нужно отделить их *двумя* пустыми строками.

Руководство по стилю оформления кода на Python: https://pep8.org/ (англ.), http://pep8.ru/doc/pep8/ (рус.)

Эту проблему тоже легко исправить: *вставьте еще одну пустую строку между функциями.* Давайте внесем правки и повторим проверку.

Проверка на совместимость с PEP 8

После внесения правок в код `vsearch.py` содержимое файла выглядит так.

```
def search4vowels(phrase: str) -> set:
    """Return any vowels found in a supplied phrase."""
    vowels = set('aeiou')
    return vowels.intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Return a set of the 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Версия «vsearch.py», совместимая с PEP 8.

Проверка этой версии кода с помощью плагина **pep8** для **pytest** подтвердила, что теперь у нас нет проблем совместимости с PEP 8. Вот что мы увидели на компьютере (с *Windows*):

Зеленая строка на экране — это хорошо! В коде нет проблем совместимости с PEP 8. ☺

```
C:\Windows\system32\cmd.exe

E:\_NewBook\ch04>py.test --pep8 vsearch.py
============================= test session starts =============================
platform win32 -- Python 3.5.0, pytest-2.8.7, py-1.4.31, pluggy-0.3.1
rootdir: E:\_NewBook\ch04, inifile:
plugins: pep8-1.0.6
collected 1 items

vsearch.py .

========================== 1 passed in 0.06 seconds ===========================

E:\_NewBook\ch04>
```

Проверка на совместимость с PEP 8 — это хорошо

Если вы не понимаете, зачем это делать (особенно бороться с пробелами), подумайте, для чего вообще нужна PEP 8. В документации сказано, что такое оформление кода *улучшает его читаемость, а код гораздо чаще читают уже после того, как он написан*. Если вы оформляете код в стандартном стиле, его становится проще читать, потому что он похож на то, что привыкли видеть другие программисты. Единообразие — отличная идея.

Далее весь код в книге будет соответствовать PEP 8. Можете проверить.

Это конец обзора pytest. До встречи в главе 5.

Код из главы 4

```
def search4vowels(phrase: str) -> set:
    """Returns the set of vowels found in 'phrase'."""
    return set('aeiou').intersection(set(phrase))


def search4letters(phrase: str, letters: str='aeiou') -> set:
    """Returns the set of 'letters' found in 'phrase'."""
    return set(letters).intersection(set(phrase))
```

Код из модуля «vsearch.py» с двумя функциями: «search4vowels» и «search4letters».

Файл «setup.py», помогающий превратить модуль в дистрибутив.

```
from setuptools import setup

setup(
    name='vsearch',
    version='1.0',
    description='The Head First Python Search Tools',
    author='HF Python 2e',
    author_email='hfpy2e@gmail.com',
    url='headfirstlabs.com',
    py_modules=['vsearch'],
)
```

```
def double(arg):
    print('Before: ', arg)
    arg = arg * 2
    print('After:  ', arg)


def change(arg: list):
    print('Before: ', arg)
    arg.append('More data')
    print('After:  ', arg)
```

Модуль «mystery.py», который использовали Роман и Юлия в споре друг с другом. К счастью, загадка разгадана, и они снова отличные приятели. ☺

Возвращение в реальный мир

Вы уже знаете Python достаточно, чтобы быть опасными.

Четыре главы книги освоены, и сейчас вы в состоянии продуктивно использовать Python во многих областях применения (хотя многое еще предстоит узнать). Вместо того чтобы исследовать эти области, в этой и последующих главах мы изучим разработку веб-приложений — в этом Python особенно силен. Попутно вы еще больше узнаете о Python. Однако вначале позвольте кратко подытожить, что вы уже знаете о Python.

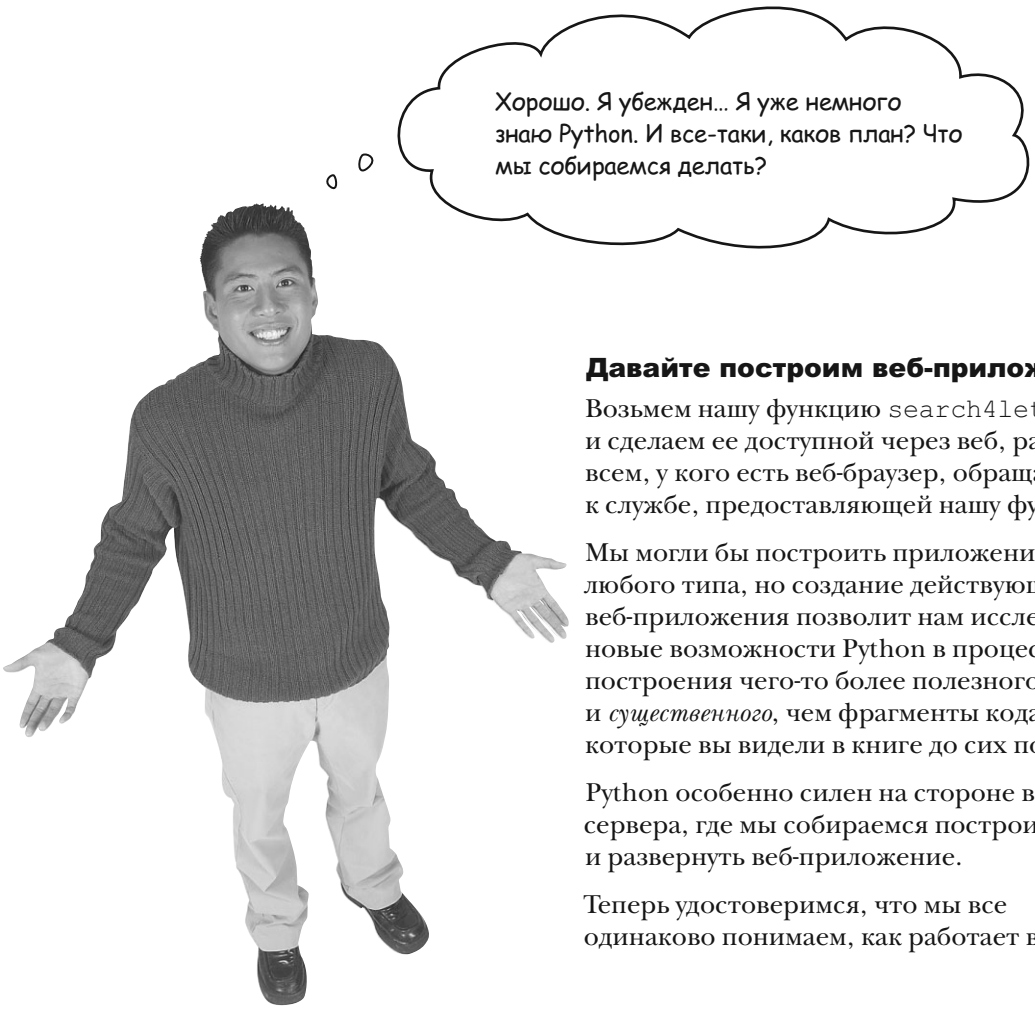
Python: что вы уже знаете

После четырех глав можно остановиться и посмотреть, что вы уже знаете.

КОНТРОЛЬНЫЙ СПИСОК

- `IDLE`, встроенная в Python `IDE`, используется для экспериментов с кодом на Python. Позволяет выполнять небольшие фрагменты кода (по одной инструкции в каждом) и большие многострочные программы, написанные в текстовом редакторе `IDLE`. Вы научились запускать файлы с кодом на Python не только в `IDLE`, но и непосредственно из командной строки своей операционной системы, используя команду `py -3` (в Windows) или `python3` (во всех остальных).
- Python поддерживает элементы данных, хранящие единственное значение, такие как целые числа и строки, а также логические `True` и `False`.
- Вы исследовали использование встроенных структур данных: списков, словарей, множеств и кортежей. Узнали, что можно создавать сложные структуры данных, комбинируя встроенные типы разными способами.
- Вы использовали широкий набор инструкций Python, включая `if`, `elif`, `else`, `return`, `for`, `from` и `import`.
- В Python имеется богатая стандартная библиотека, и вы увидели, как работают следующие модули: `datetime`, `random`, `sys`, `os`, `time`, `html`, `pprint`, `setuptools` и `pip`.
- Помимо стандартной библиотеки в Python имеется коллекция удобных встроенных функций. Вот некоторые из функций, с которыми вы работали: `print`, `dir`, `help`, `range`, `list`, `len`, `input`, `sorted`, `dict`, `set`, `tuple` и `type`.
- Python поддерживает все обычные операторы плюс еще некоторые. В число тех, что вы уже видели, входят: `in`, `not in`, `+`, `—`, `=` (присваивание), `==` (проверка на равенство), `+=` и `*`.
- Помимо нотации с квадратными скобками для работы с элементами последовательностей (то есть `[]`) Python поддерживает нотацию **срезов**, которая позволяет определять **начальный** и **конечный** индексы, а также величину **шага**.
- Вы научились создавать собственные функции на языке Python, используя выражение `def`. Функции в Python могут принимать произвольное количество аргументов, а также возвращать значение.
- Несмотря на возможность заключать строки в одиночные или двойные кавычки, соглашение Python (задокументированное в **PEP 8**) предлагает выбрать один стиль и придерживаться его. В этой книге мы решили заключать все строки в одиночные кавычки, кроме строк, которые сами содержат символ одиночной кавычки, в этом случае мы используем двойные кавычки (как исключение, особенный случай).
- Также поддерживаются строки, заключенные в тройные кавычки, и вы видели, как они используются для добавления строк документации в ваши собственные функции.
- Связанные функции можно объединять в модули. Модули — это основа повторного использования кода в Python, и вы видели, как модуль `pip` (включенный в стандартную библиотеку) позволяет единообразно управлять установкой модулей.
- Коль скоро речь зашла о единообразии: вы узнали, что в Python **все является объектом**, а это гарантирует (насколько возможно), что все работает в полном соответствии с ожиданиями. Такая концепция действительно окупается, когда вы начинаете определять пользовательские объекты, используя классы, с которыми мы познакомим вас в следующей главе.

Давайте что-нибудь построим



Давайте построим веб-приложение.

Возьмем нашу функцию `search4letters` и сделаем ее доступной через веб, разрешив всем, у кого есть веб-браузер, обращаться к службе, предоставляющей нашу функцию.

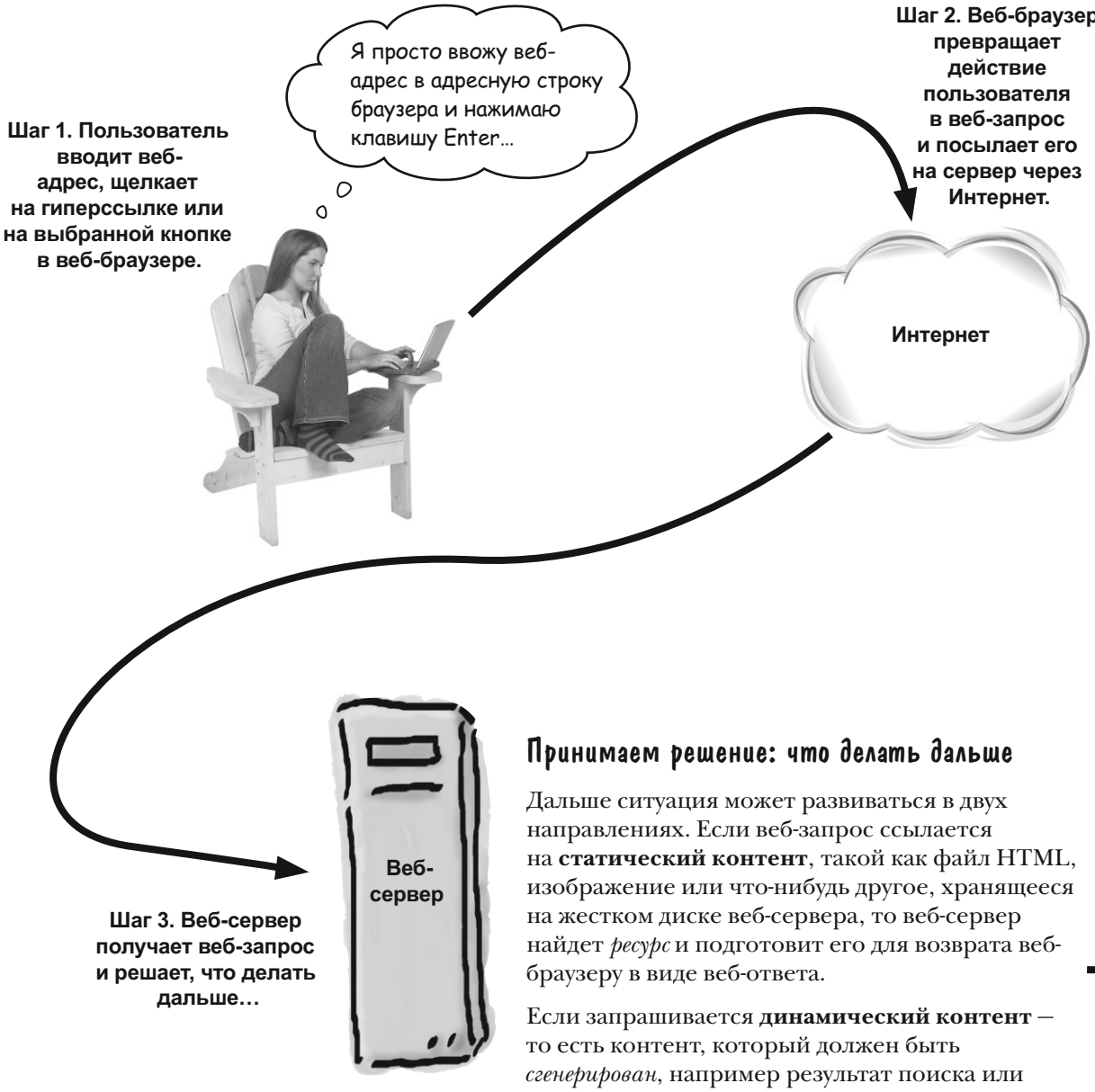
Мы могли бы построить приложение любого типа, но создание действующего веб-приложения позволит нам исследовать новые возможности Python в процессе построения чего-то более полезного и *существенного*, чем фрагменты кода, которые вы видели в книге до сих пор.

Python особенно силен на стороне веб-сервера, где мы собираемся построить и развернуть веб-приложение.

Теперь удостоверимся, что мы все одинаково понимаем, как работает веб.

Веб-приложения под лупой

Чем бы вы ни занимались в веб, все сводится к *запросам* и *ответам*. В результате некоторых действий пользователя веб-браузер посылает веб-серверу **веб-запросы**. Веб-сервер формирует **веб-ответ** (или *отклик*) и возвращает его веб-браузеру. Весь этот процесс можно представить в виде пяти следующих шагов.



Принимаем решение: что делать дальше

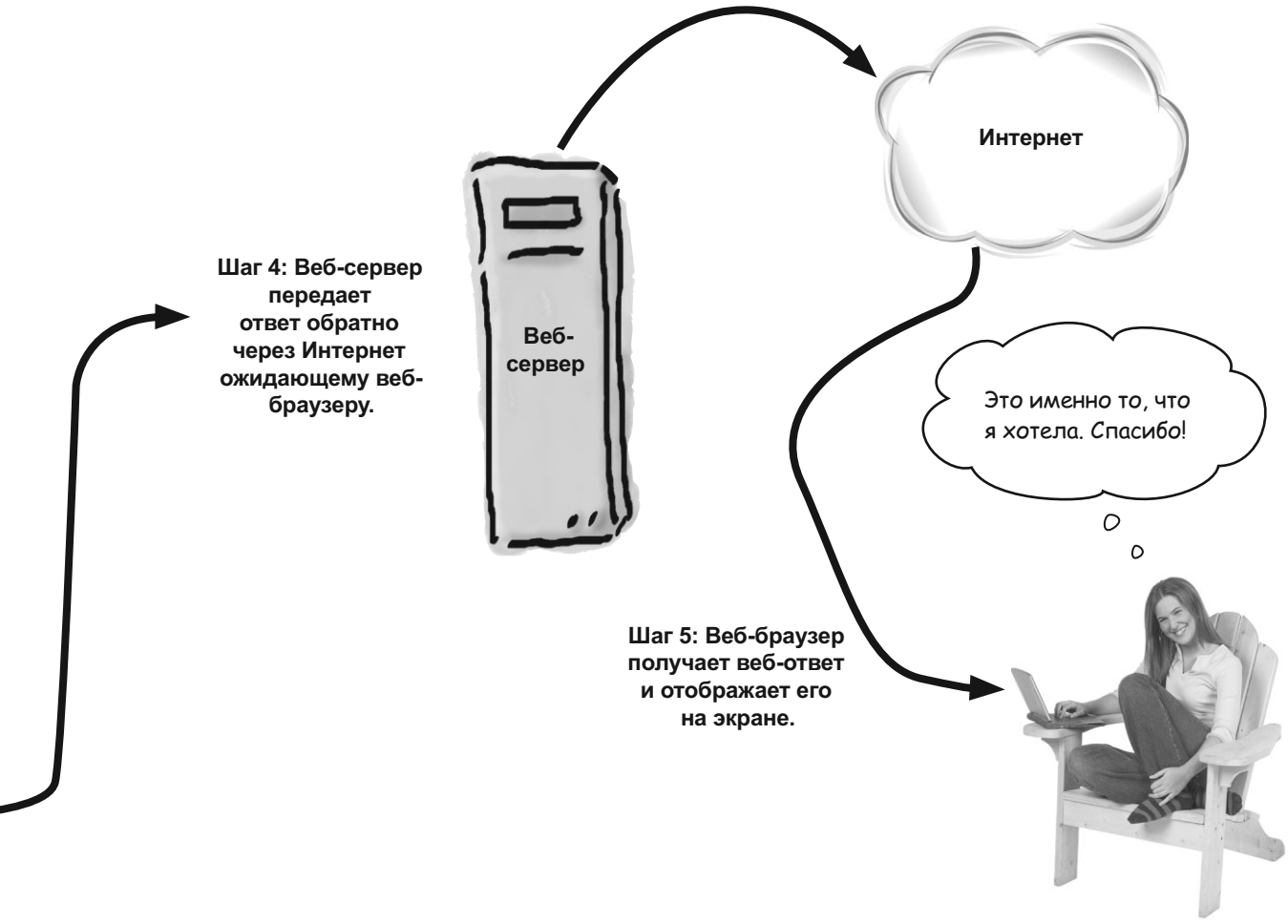
Дальше ситуация может развиваться в двух направлениях. Если веб-запрос ссылается на **статический контент**, такой как файл HTML, изображение или что-нибудь другое, хранящееся на жестком диске веб-сервера, то веб-сервер найдет *ресурс* и подготовит его для возврата веб-браузеру в виде веб-ответа.

Если запрашивается **динамический контент** — то есть контент, который должен быть *сгенерирован*, например результат поиска или текущее содержимое корзины онлайн-магазина, то веб-сервер запустит некоторый код, чтобы создать веб-ответ.

Множество (возможное) вариантов шага 3

На практике шаг 3 может включать множество стадий, в зависимости от того, что должен сделать веб-сервер для создания ответа. Очевидно, что если серверу достаточно найти статический контент и вернуть его браузеру, этапы не слишком сложные: задача сводится к простому чтению файлов с жесткого диска веб-сервера.

Если требуется сгенерировать динамический контент, веб-сервер выполняет такие этапы, как запуск кода, захват вывода программы и подготовка на его основе веб-ответа для отправки ожидающему веб-браузеру.



Чего мы хотим от нашего веб-приложения?

Как же заманчиво *просто начать программировать*, но давайте подумаем о том, как будет работать наше веб-приложение.

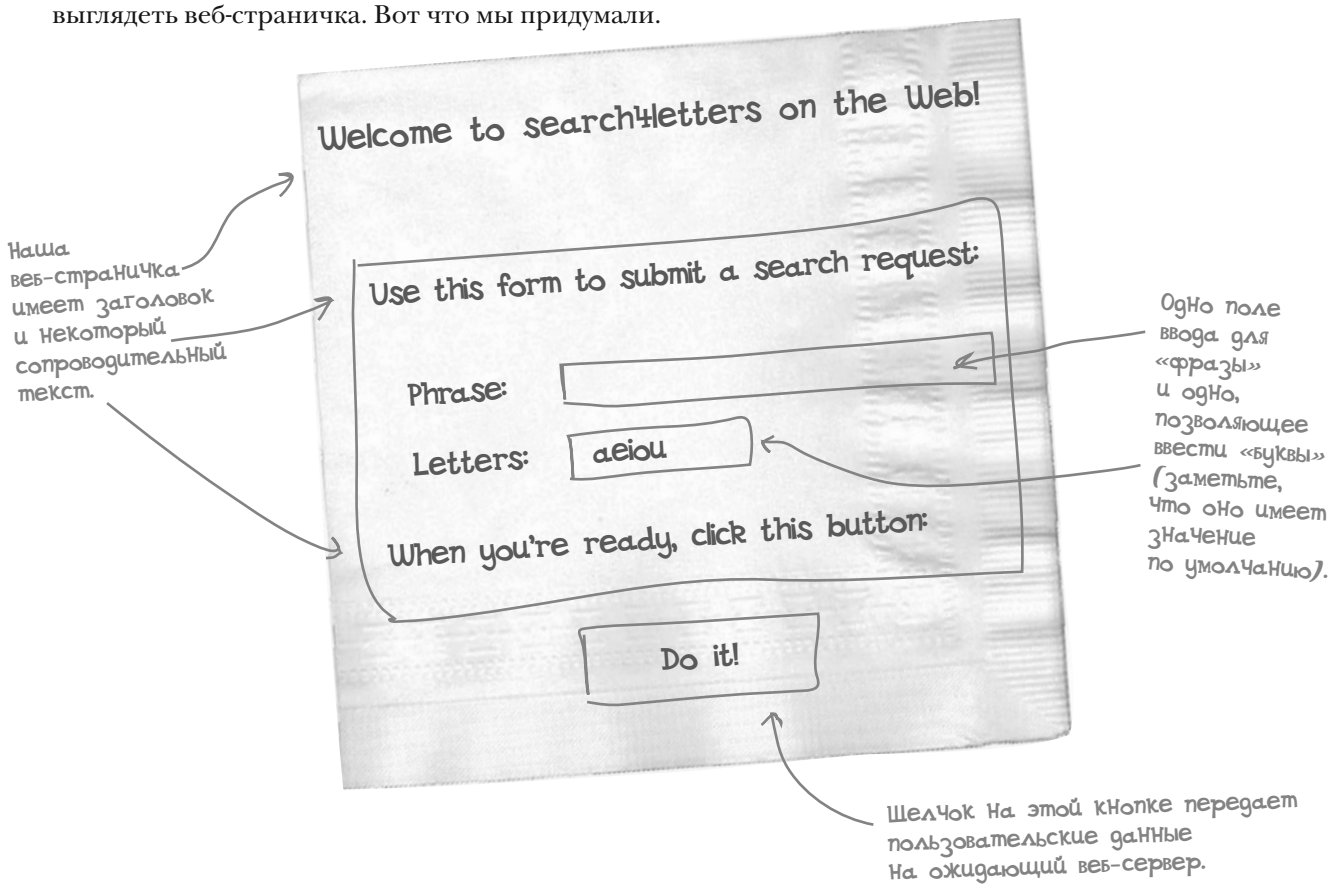
Пользователь будет взаимодействовать с веб-приложением, используя свой любимый веб-браузер. Чтобы получить доступ к услуге, он должен ввести URL веб-приложения в адресную строку браузера. Веб-страница, которая появится в браузере, предложит пользователю ввести аргументы для функции `search4letters`. После их ввода пользователь должен нажать кнопку, чтобы увидеть результаты.

Согласно определению в строке `def`, последняя версия функции `search4letters` ожидает не менее одного и не более двух аргументов: фразу и буквы, которые нужно найти во фразе. Как вы помните, аргумент `letters` необязателен (по умолчанию он принимает значение `'aeiou'`).

Строка «def» определения функции «search4letters», которая сообщает, что она принимает не менее одного и не более двух аргументов.

```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

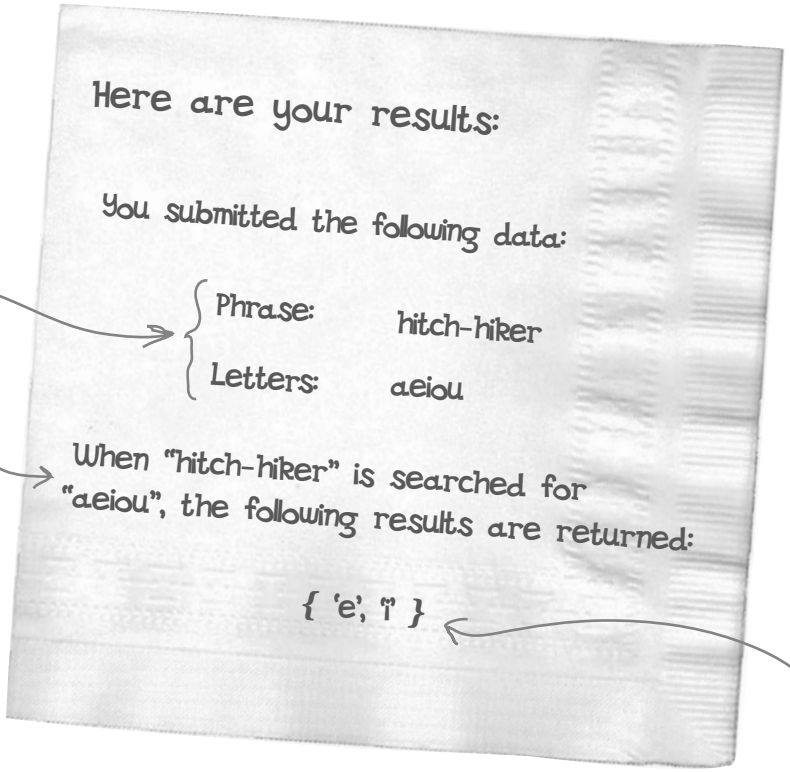
Возьмем бумажную салфетку и попробуем изобразить, как должна выглядеть веб-страничка. Вот что мы придумали.



Что произойдет на веб-сервере?

Когда пользователь щелкнет на кнопке **Do it!**, браузер перешлет данные на ожидающий веб-сервер, который извлечет значения `phrase` и `letters`, а затем вызовет функцию `search4letters` от имени ожидающего пользователя.

Все результаты выполнения функции возвращаются веб-браузеру пользователя в составе другой веб-странички, которую мы опять набросаем на бумажной салфетке (см. ниже). Предположим, что пользователь ввел фразу «hitch-hiker» и оставил строку с буквами для поиска по умолчанию `'aeiou'`. Тут показано, как может выглядеть полученная веб-страничка.



Переданные данные возвращены пользователю.

Результат, который вернула «search4letters», также присутствует на страничке.

Что нам надо, чтобы начать работать?

Помимо имеющихся знаний о Python, для создания веб-приложения, работающего на стороне сервера, вам понадобится **фреймворк веб-приложений**. Он реализует набор общих основополагающих технологий, на которых вы сможете построить свое веб-приложение.

Хотя Python позволяет сконструировать все необходимое с самого начала, делать это было бы безумием. Другие программисты уже потратили время, чтобы создать для вас веб-фреймворки. Python предлагает много вариантов. Однако мы не будем мучиться вопросом выбора, а просто возьмем популярный фреймворк *Flask* и двинемся дальше.

Давайте установим Flask

Как мы узнали в главе 1, стандартная библиотека Python поставляется с *батарейками в комплекте*. Однако иногда необходимы специальные сторонние модули, не являющиеся частью стандартной библиотеки. Сторонние модули импортируются в программу на Python по мере необходимости. В отличие от модулей стандартной библиотеки, сторонние модули должны быть установлены *до того*, как будут импортированы и использованы. Flask — это один из сторонних модулей.

Как упоминалось в предыдущей главе, сообщество Python поддерживает централизованный веб-сайт для сторонних модулей, называющийся **PyPI** (сокращение для *Python Package Index* — каталог пакетов для Python), на котором находится последняя версия Flask (так же как и многие другие проекты).

Найдем PyPI на pypi.python.org

Вспомните, как мы использовали `pip` для установки нашего модуля `vsearch` в Python ранее в этой книге (`pip` также работает с PyPI). Если вы знаете название нужного модуля, то можете использовать `pip` для установки любого модуля, размещенного на PyPI, прямо из окружения Python.

Установка Flask из командной строки с помощью pip

Если вы работаете в *Linux* или *Mac OS X*, введите в окно терминала следующую команду.

```
$ sudo -H python3 -m pip install flask
```

Используйте эту команду в Mac OS X и Linux.

Если вы работаете в *Windows*, откройте командную строку с правами администратора (щелкнув правой кнопкой мыши на файле `cmd.exe` и выбрав в контекстном меню пункт *Run as Administrator (Запуск от имени администратора)*), а затем выполните следующую команду.

Обратите внимание: регистр символов важен. Это «f» в нижнем регистре для «flask».

```
C:\> py -3 -m pip install flask
```

Используйте эту команду в Windows.

Эта команда (независимо от операционной системы) подключится к веб-сайту PyPI, затем загрузит и установит модуль **Flask** и четыре других модуля, от которых зависит модуль Flask: **Werkzeug**, **MarkupSafe**, **Jinja2** и **itsdangerous**. Не беспокойтесь (сейчас) о назначении дополнительных модулей; просто удостоверьтесь, что они корректно установились. Если установка прошла успешно, в конце вывода, сгенерированного `pip`, вы увидите сообщение, похожее на то, что приводится ниже. Заметьте, что вывод содержит около дюжины строк, примерно таких:

```
...
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11 flask-0.10.1
itsdangerous-0.24
```

Эти версии модулей были текущими на момент написания книги.

Если появилось сообщение «`Successfully installed...`» (успешно установлено...), убедитесь, что компьютер подключен к Интернету и вы ввели команду для операционной системы *в точности так*, как мы показывали выше. Не волнуйтесь, если номера версий модулей, установленных у вас, отличаются от наших (модули постоянно обновляются, и зависимости также могут поменяться). Поскольку версии, которые вы устанавливаете, *по меньшей мере* не ниже тех, что показаны выше, все в порядке.

Как работает Flask?

Flask предоставляет коллекцию модулей, которые помогают создавать серверные веб-приложения. Технически это *микро*-веб-фреймворк, который поддерживает минимальный набор технологий, необходимых для выполнения этой задачи. Flask не настолько полнофункциональный, как некоторые из его конкурентов (таких как **Django**, мать всех веб-фреймворков в Python), но он маленький, легковесный и простой в использовании.

Так как наши требования не слишком велики (у нас только две веб-странички), возможностей веб-фреймворка Flask для нас более чем достаточно.

Для умников

Django — это чрезвычайно популярный в сообществе Python фреймворк веб-приложений. Он имеет мощные встроенные средства администрирования, которые могут сделать работу с большим количеством веб-приложений очень гибкой. Для того, что мы делаем сейчас, использовать **Django** — все равно что стрелять из пушки по воробьям, поэтому мы выбрали простой и легковесный **Flask**.

Проверка готовности Flask

Ниже приведен код простейшего веб-приложения Flask, с помощью которого мы проверим готовность фреймворка Flask к использованию.

В текстовом редакторе создайте новый файл, введите код, приведенный ниже, и сохраните его как `hello_flask.py` (мы сохранили файл в папке `webapp`, но вы можете сохранить его в любой другой папке).

Готовый код

Это «hello_flask.py».

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Введите этот код в точности как показано здесь... что это значит, мы узнаем через минутку.

Запуск Flask из командной строки вашей операционной системы

Не поддавайтесь искушению запустить код, использующий Flask, в интерактивной оболочке IDLE, потому что она для этого не предназначена. IDLE хороша для экспериментов с небольшими фрагментами кода, но для запуска приложений лучше вызывать интерпретатор непосредственно из командной строки операционной системы. Давайте сделаем это и посмотрим, что случится.

Не используйте IDLE для запуска этого кода.

Первый запуск веб-приложения Flask

Если вы работаете в *Windows*, откройте командную строку в папке с файлом программы `hello_flask.py`. (Подсказка: если папка открыта в *Проводнике*, нажмите клавишу Shift и, удерживая ее, щелкните правой кнопкой и выберите в контекстном меню пункт *Open command window here (Запуск командной строки здесь)*.) Когда откроется окно командной строки *Windows*, введите следующую команду, чтобы запустить приложение Flask.

Мы сохранили код в папке с названием «webapp».

```
C:\webapp> py -3 hello_flask.py
```

Просим интерпретатор Python выполнить код из «hello_flask.py».

Если вы работаете в *Mac OS X* или *Linux*, введите следующую команду в окне терминала. Учтите, что эта команда должна выполняться в папке с файлом программы `hello_flask.py`.

```
$ python3 hello_flask.py
```

Вне зависимости от операционной системы, с этого момента Flask возьмет управление на себя и будет выводить сообщения о текущем состоянии на экран каждый раз, когда встроенный веб-сервер выполнит какую-либо операцию. Сразу после запуска веб-сервер Flask подтверждает, что он запущен и готов к обслуживанию веб-запросов, поступающих на тестовый веб-адрес (`127.0.0.1`) в порт протокола (`5000`).

Если вы видите это сообщение, все хорошо.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Веб-сервер Flask готов и ожидает. *Что теперь?* Попробуем обратиться к веб-серверу, используя веб-браузер. Откройте браузер, который вам больше по вкусу, и введите URL из сообщения, которое вывел веб-сервер Flask в момент запуска.

Это адрес запущенного веб-приложения. Введите его в точности так, как показано здесь.

```
http://127.0.0.1:5000/
```

Через мгновение в окне браузера сообщение должно появиться «Hello world from Flask!» («Привет миру из Flask!») из `hello_flask.py`. Взгляните на окно терминала, где было запущено веб-приложение... там должно появиться новое сообщение:

Ага! Что-то произошло.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
```

Для умников

Подробное описание, что такое **номер порта протокола**, выходит за рамки этой книги. Однако если вы хотите это знать, начните с чтения этой страницы:

https://en.wikipedia.org/wiki/Port_(computer_networking)

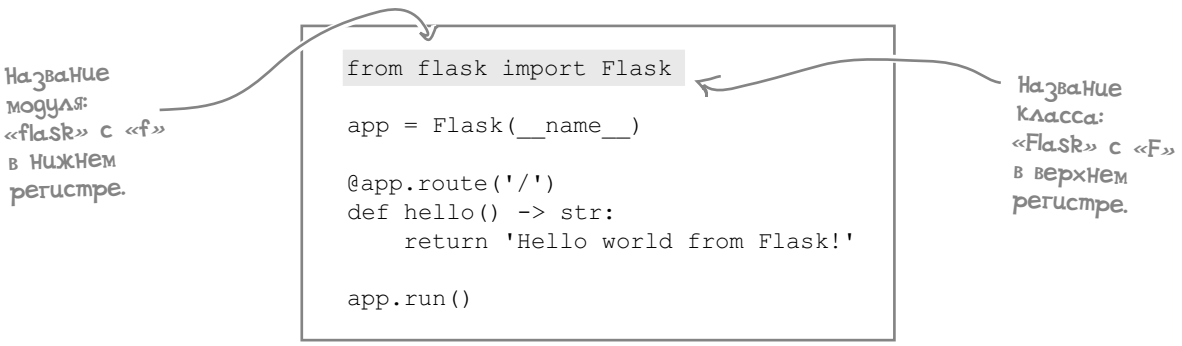
Вот что произошло (шаг за шагом)

В дополнение к тому, что Flask вывел в терминал строку с информацией о состоянии, ваш веб-браузер сейчас показывает ответ веб-сервера. Вот как сейчас выглядит наш браузер (*Safari* в *Mac OS X*).

В ответ на попытку открыть в браузере страницу с URL, указанном в сообщении о состоянии веб-приложения, сервер ответил сообщением «Hello world from Flask!».

Хотя наше веб-приложение содержит всего шесть строк кода, в нем происходит много интересного. Пройдемся по нему шаг за шагом и посмотрим, что происходит. Все остальное мы планируем сделать, основываясь на этих шести строках кода.

Первая строка импортирует класс `Flask` из модуля `flask`.

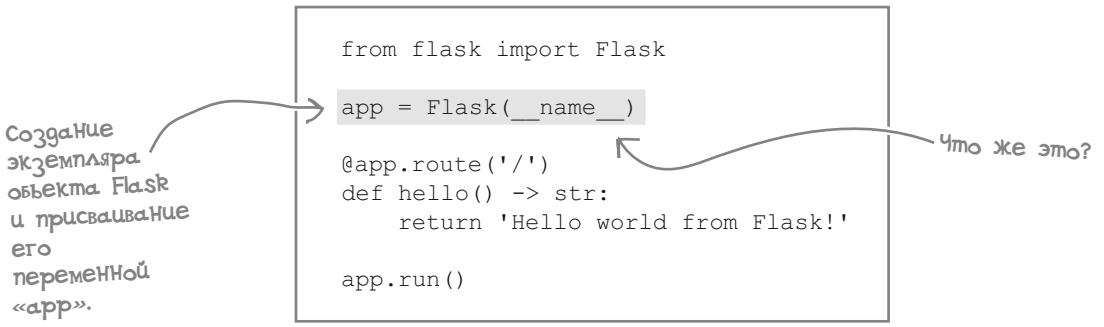


Помните, как мы обсуждали альтернативные способы импортирования?

Вы могли написать здесь `import flask` и затем ссылаться на класс `Flask` как `flask.Flask`, но использование варианта `from` инструкции `import` в данном случае предпочтительнее, потому что читать код `flask.Flask` сложнее, чем просто `Flask`.

Создание объекта веб-приложения Flask

Вторая строка создает объект типа `Flask` и присваивает его переменной `app`. Она выглядит просто, если бы не странный аргумент `__name__`.



Значение ***__name__*** определяется интерпретатором Python и, когда используется где-либо в программном коде, устанавливается равным имени текущего активного модуля. Оказывается, классу `Flask` нужно знать текущее значение `__name__`, когда создается новый объект `Flask`, поэтому оно должно быть передано в аргументе, что мы и сделали здесь (хотя это и выглядит *странно*).

Эта одна строка кода, несмотря на ее лаконичность, делает потрясающе много. Фреймворк `Flask` скрывает множество деталей, позволяя вам сконцентрироваться на определении желаемого поведения в ответ на веб-запрос, поступающий на ожидающий сервер. Мы сделаем это прямо в следующей строке.

Для умников

Обратите внимание, что `__name__` состоит из двух символов подчеркивания, слова «name» и еще двух символов подчеркивания. Мы называем их «двойное подчеркивание», когда используем в именах в качестве префикса и суффикса. Вы будете часто видеть это соглашение об именовании в вашем путешествии по Python, и вместо многословной фразы «двойное подчеркивание, имя, двойное подчеркивание» программисты на Python, руководствуясь здравым смыслом, говорят: «имя с двойными подчеркиваниями», что является **сокращением для того же понятия**. Двойные подчеркивания часто и много используются в Python, вы увидите множество примеров других имен с двойными подчеркиваниями и их использования в оставшейся части книги.

Наравне с двойными подчеркиваниями, также существует соглашение об именах переменных, начинающихся с одного символа.

Декорирование функции URL

Следующая строка кода знакомит вас с новой синтаксической конструкцией Python: **декораторами**. Декоратор функции, который мы видим в этой строке, настраивает поведение существующей функции *без* изменения кода самой функции (то есть функция становится декорированной).

Перечитайте предыдущее предложение несколько раз.

В сущности, декораторы позволяют взять существующий код и добавить к нему дополнительное поведение. Хотя декораторы применимы не только к функциям, но и к классам, в основном их применяют к функциям и называют **декораторами функций**.

Посмотрим на декоратор функции в коде нашего веб-приложения. Его легко обнаружить, поскольку он начинается с символа `@`.

Для умников

Синтаксис декораторов в Python берет начало из синтаксиса аннотаций в Java, а также из мира функционального программирования.

Вот декоратор функции, который — как все декораторы — начинается с символа @.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Это URL.

Вы можете создавать собственные декораторы функций (об этом рассказано в следующей главе), но сейчас мы сосредоточимся только на их использовании. В Python есть много встроенных декораторов и много сторонних модулей (таких как Flask), предоставляющих свои декораторы для определенных целей (`route` — это один из них).

Декоратор `route` из Flask доступен в коде нашего веб-приложения через переменную `app`, созданную в предыдущей строке.

Декоратор `route` позволяет связать веб-путь URL с существующей функцией на Python. В этом случае URL «/» связывается с функцией `hello`, которая определяется в следующей строке. Декоратор `route` организует вызов указанной функции веб-сервером Flask, когда тот получает запрос с URL «/». Затем декоратор `route` ожидает вывод от декорированной функции и передает его серверу, который возвращает его ожидающему веб-браузеру.

Не важно, как Flask (и декоратор `route`) творит описанное «волшебство». Важно, что Flask делает это для вас, и вам остается только написать функцию, которая создаст необходимый вывод. Об остальном позаботятся Flask и декоратор `route`.

Декоратор настраивает поведение существующей функции (без изменения кода).

Запуск функций веб-приложения

Функция, декорируемая декоратором `route`, начинается в следующей строке. В нашем веб-приложении это функция `hello`, которая делает только одно: возвращает сообщение «Hello world from Flask!».

Это обычная функция Python, возвращающая строку (обратите внимание на аннотацию «-> str»).

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Последняя строка в программе предлагает объекту `Flask` в переменной `app` запустить веб-сервер, вызывая метод `run`.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Просим веб-приложение начать выполнение.

С этого момента Flask запускает встроенный веб-сервер и выполняет в нем код веб-приложения. На любые запросы с URL «/» веб-сервер ответит сообщением «Hello world from Flask!», а на запросы с любыми другими URL — сообщением об ошибке 404 «Resource not found» (ресурс не найден). Чтобы увидеть обработку ошибки в действии, введите этот URL в адресную строку вашего браузера.

http://127.0.0.1:5000/doesthiswork.html

Ваш браузер покажет сообщение «Not Found» (Не найдено), и веб-приложение выведет в окне терминала соответствующее сообщение с информацией о состоянии.

Этот URL не существует: 404!

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:30:26] "GET /doesthiswork.html HTTP/1.1" 404 -
```

Сообщение у вас может немного отличаться. Но пусть вас это не беспокоит.

Размещение функциональности в веб

Отложим в сторону тот факт, что мы только что построили работающее веб-приложение, написав всего шесть строк кода, и рассмотрим, что сделал Flask. Он предоставил механизм, позволяющий взять любую существующую функцию на Python и показать ее вывод в веб-браузере.

Чтобы расширить возможности веб-приложения, достаточно просто выбрать URL и написать соответствующую строку с декоратором `@app.route` перед функцией, выполняющей фактическую работу. Давайте сделаем это, используя функцию `search4letters` из предыдущей главы.

Заточите карандаш

Добавим в `hello_flask.py` обработку второго URL: `/search4`. Напишите код, который свяжет этот URL с функцией `do_search`, вызывающей функцию `search4letters` (из модуля `vsearch`). Затем организуйте в функции `do_search` возврат результата, получаемого при поиске строки символов `'eiru,!'` во фразе: «life, the universe, and everything!».

Ниже приведен код с зарезервированными строками для нового кода, который вам нужно написать. Задача — дописать недостающий код.

Подсказка: функция `search4letters` возвращает множество. Его нужно преобразовать в строку вызовом встроенной функции `str` и только потом вернуть ожидающему веб-браузеру, так как он ожидает текстовые данные, а не множество Python.

Надо что-нибудь импортировать?

```
from flask import Flask
.....
.....

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'
```

Добавьте второй декоратор.

```
.....
.....
.....
.....

app.run()
```

Добавьте код функции «do_search».

Заточите карандаш
Решение

Требовалось добавить в `hello_flask.py` обработку второго URL: `/search4`, связав его с функцией `do_search`, которая сама вызывает функцию `search4letters` (из модуля `vsearch`). Нужно было организовать в функции `do_search` возврат результата, получаемого при поиске строки символов `'eiru,!'` во фразе: «life, the universe, and everything!».

Ниже приведен код с зарезервированными строками для нового кода, который вам нужно написать. Задача заключалась в добавлении недостающего кода.

Совпадает ли ваш код с нашим?

Прежде чем вызвать функцию «search4letters», нужно импортировать ее из модуля «vsearch».

```
from flask import Flask

from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

Второй декоратор устанавливает URL «/search4».

Функция «do_search» вызывает «search4letters» и возвращает результат в виде строки.

Для проверки новой функциональности перезапустите веб-приложение, поскольку сейчас оно запущено со старой версией кода. Для этого вернитесь в окно терминала, а затем одновременно нажмите клавиши Ctrl и C. Веб-приложение будет завершено и вернет управление командной строке операционной системы. Нажмите клавишу со стрелкой вверх, чтобы вернуться к предыдущей команде (которая до этого запускала `hello_flask.py`), и нажмите Enter. В окне вновь появится начальное сообщение, подтверждающее, что обновленное веб-приложение ожидает запросов.

Останавливаем веб-приложение...

...затем перезапускаем его.

```
$ python3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:30:26] "GET /doesthiswork.html HTTP/1.1" 404 -
^C
$ python3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Приложение запущено и выполняется.

Пробная поездка

Поскольку вы не изменили код, связанный с URL «/» по умолчанию, он все еще работает, возвращая сообщение «Hello world from Flask!».

Но если вы введете ***http://127.0.0.1:5000/search4*** в адресную строку браузера, то увидите результат вызова `search4letters`.

Результат вызова «search4letters». Конечно, этот вывод не вызывает бурный восторг, но доказывает, что при обращении к URL «/search4» вызывается функция и возвращается результат.

Это не глупые вопросы

В: Я немного запутался с частями URL — `127.0.0.1` и:`5000` — используемыми для доступа к веб-приложению. Какую роль они играют?

О: Сейчас вы проверяете веб-приложение на своем компьютере, который подключен к Интернету и имеет уникальный IP-адрес. Несмотря на этот факт, Flask не использует ваш IP-адрес и подключает тестовый веб-сервер к **внутреннему адресу**: `127.0.0.1`, также известному как `localhost`. Оба являются коротким обозначением «моего компьютера, независимо от фактического IP-адреса». В своем веб-браузере (также на вашем компьютере) для подключения к своему веб-серверу Flask вы должны указать адрес своего веб-приложения, то есть `127.0.0.1`. Это стандартный IP-адрес, зарезервированный как раз для этой цели.

Часть:`5000` в URL определяет **номер порта протокола**, который прослушивает ваш веб-сервер.

Как правило, веб-серверы прослушивают порт `80`, стандартный для Интернета и не требующий явного указания. Вы можете ввести `oreilly.com:80` в адресную строку браузера, и этот прием сработает, но так никто не делает, потому что адреса `oreilly.com` вполне достаточно (так как: `80` подразумевается).

В процессе разработки веб-приложений очень редко используется порт `80` (зарезервированный для действующих серверов), поэтому большинство фреймворков выбирают другой порт для запуска. Порт с номером `8080` также весьма популярен, но для тестирования Flask использует номера порта `5000`.

В: Могу я использовать какой-нибудь другой порт, отличный от `5000`, когда проверяю и запускаю свое веб-приложение на Flask?

О: Да, `app.run()` позволяет определить любой другой номер порта. Но сейчас, если только у вас нет очень веской причины для его изменения, оставьте для Flask номер порта по умолчанию, значение которого равно `5000`.

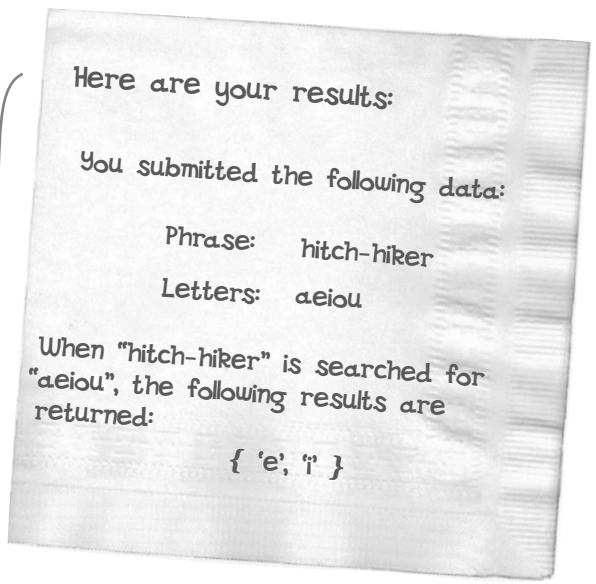
Вспомним, что мы пытались построить

Нашему веб-приложению требуется веб-страничка, которая принимает ввод, и другая, отображающая результат передачи введенных значений функции `search4letters`. Текущий код нашего веб-приложения делает далеко не все, что требуется, но образует основу для построения того, что нам нужно.

Внизу слева приведена копия нашего текущего кода, а справа — копия уже знакомой вам «салфеточной спецификации». Мы определили, где в коде можно реализовать функциональность с каждой салфетки.

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters( ... ))

app.run()
```

Обратите внимание: чтобы все поместилось, мы не показали строку целиком.



Вот план

Изменим функцию `hello`, чтобы она возвращала HTML-форму. Затем изменим функцию `do_search`, чтобы перед вызовом функции `search4letters` она принимала введенные данные из формы и возвращала результаты в виде другой веб-странички.

Построение HTML-формы

Требуемая HTML-форма не очень сложная. Кроме описательного текста, форма содержит два поля ввода и кнопку.

Но... что, если мы новички в этих HTML-штуках?

Не паникуйте, если все эти разговоры о HTML-формах, полях ввода и кнопках волнуют вас. У нас есть то, что вы ищете: второе издание книги «Изучаем HTML, XHTML и CSS» (*Head First HTML and CSS*) — лучшее введение для тех, кому требуется быстро ознакомиться с этими технологиями (или освежить память).

Если мысль о том, чтобы отложить эту книгу ради изучения языка разметки HTML, кажется вам малопривлекательной, заметьте, что мы предоставляем весь код HTML, который требуется для работы с примерами, и избавляем вас от необходимости глубоко изучать HTML. Знакомство с HTML — желательное, но не обязательное требование (в конце концов, книга посвящена Python, а не HTML).

Создайте код HTML, а затем отправьте его в браузер

Как обычно, одно и то же можно сделать несколькими способами, и когда наступит время создать HTML-текст в вашем Flask веб-приложении, у вас есть выбор.

Я предпочитаю определять разметку HTML в виде **длинных строк**, которые затем использую в коде на Python, возвращая эти строки по необходимости. Благодаря этому в моем коде есть все необходимое, и я имею полный контроль... вот как я это делаю. Что тебе не нравится, Анна?

Да, Макс, подход с включением всей разметки HTML в код работает, но это решение не масштабируемо. По мере развития веб-приложения код со встроенной HTML-разметкой превращается в кашу... и ее сложно передать веб-дизайнеру для улучшения. Также нелегко повторно использовать фрагменты HTML. Поэтому в веб-приложениях я всегда использую **механизм шаблонов**. Это требует немного больше работы вначале, но через некоторое время я обнаруживаю, что использование шаблонов окупается...

Анна права. Шаблоны существенно упрощают обслуживание разметки HTML, если сравнивать с подходом Макса. Что ж, займемся шаблонами на следующей странице.

Шаблоны под лупой

Механизмы шаблонов позволяют программистам применять объектно-ориентированные понятия наследования и повторного использования при создании текстовых данных, таких как веб-странички.

Вид и оформление веб-сайта можно определить как высокоуровневый HTML-шаблон, известный как **базовый шаблон**, который затем наследуют другие HTML-странички. Если внести изменения в базовый шаблон, они отразятся на *всех* HTML-страничках, наследующих его.

В состав Flask входит простой и мощный механизм шаблонов, который называется *Jinja2*. Мы не стремимся описать все тонкости использования Jinja2, поэтому на двух страницах мы расскажем только самое необходимое, коротко и по существу. Более подробную информацию о возможностях Jinja2 можно посмотреть по ссылке:

http://jinja.pocoo.org/docs/dev/

Здесь представлен базовый шаблон, который мы будем использовать в веб-приложении. Мы поместили разметку HTML, общую для всех наших веб-страничек, в файл с именем `base.html`. Также мы используем специальную разметку Jinja2 для вставки содержимого из HTML-страничек, наследующих шаблон, перед их отправкой веб-браузеру. Заметьте, что разметка, которую вы видите между `{{и}}`, а также между `{%и%}`, предназначена для механизма шаблонов Jinja2: мы выделим ее, чтобы сделать более заметной.

Стандартная разметка HTML5.

Базовый шаблон.

```
<!doctype html>
<html>
    <head>
        <title>{{ the_title }}</title>
        <link rel="stylesheet" href="static/hf.css" />
    </head>
    <body>
        {% block body %}

        {% endblock %}
    </body>
</html>
```

Директива Jinja2 указывает, что значение будет вставлено перед отправкой страницы (считайте ее аргументом шаблона).

Таблица стилей определяет внешний вид веб-страничек.

Директивы Jinja2 указывают, что сюда будет подставлен блок HTML, который определяется страничкой, наследующей шаблон.

После подготовки базового шаблона его можно наследовать, используя директиву Jinja2 `extends`. В этом случае наследующие HTML-файлы должны определить разметку HTML только для именованных блоков в базовом шаблоне. Сейчас у нас имеется лишь один именованный блок: `body`.

Ниже приведена разметка для первой из наших страничек, с именем `entry.html`. Эта разметка содержит HTML-форму, с помощью которой пользователь может передать значения `phrase` и `letters`, ожидаемые веб-приложением.

Заметьте, что «шаблонная» разметка HTML из базового шаблона не повторяется в этом файле, так как директива `extends` включает всю его разметку. Нам нужно определить только разметку HTML для данной конкретной странички, и мы сделаем это, поместив ее внутрь блока Jinja2 с именем `body`.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT' width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT' value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```

Этот шаблон наследует базовый шаблон и определяет содержимое блока с именем «body».

И наконец, вот разметка для файла `results.html`, который используется для отображения результатов поиска. Этот шаблон также наследует базовый шаблон.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}", the following
results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

Как и «entry.html», этот шаблон также наследует базовый шаблон и также определяет содержимое блока с именем «body».

Обратите внимание на дополнительные аргументы, значения которых должны быть определены перед отображением.

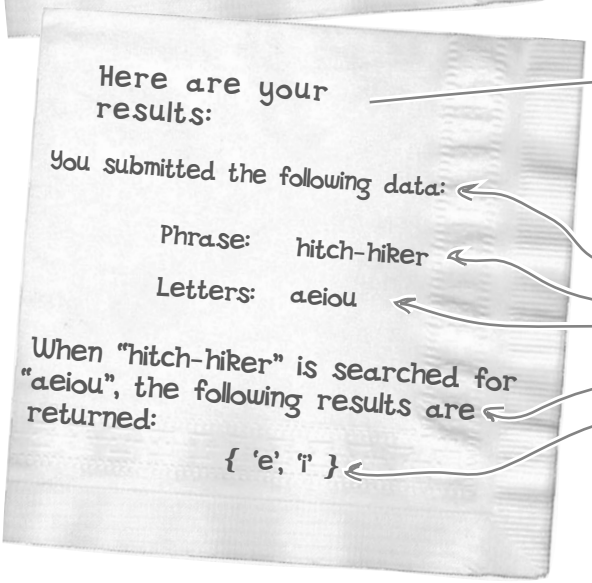
Шаблоны связаны с веб-страничками

Веб-приложение должно отображать две веб-странички, и сейчас у нас есть два шаблона, которые помогут в этом. Оба шаблона наследуют базовый шаблон и, следовательно, наследуют внешний вид базового шаблона. Осталось только отобразить странички.

Скачайте эти шаблоны (и CSS) отсюда: *http://python.itcarlow.ie/ed2/*.

Прежде чем смотреть, как Flask (вместе с Jinja2) отображает страницы, взглянем на нашу «салфеточную спецификацию» и разметку шаблона. Обратите внимание, как HTML заключается внутрь директивы `{% block%}` почти в полном соответствии с нарисованной от руки спецификацией. Главное упущение — это отсутствие заголовка в страницах, который должен заменить директиву `{{the_title}}` в ходе отображения. Рассматривайте каждое имя, заключенное в двойные фигурные скобки, как аргумент шаблона.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```




```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}",
the following results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

Не забудьте эти дополнительные аргументы.

Отображение шаблонов из Flask

Flask включает функцию с именем `render_template`, которая принимает имя шаблона со всеми необходимыми аргументами и возвращает строку с разметкой HTML. Чтобы воспользоваться функцией `render_template`, добавьте ее имя в список импорта из модуля `flask` (в начале программы), после чего вызывайте ее по мере надобности.

Но прежде переименуем файл с кодом веб-приложения (сейчас он называется `hello_flask.py`) и дадим ему более подходящее имя. Вы можете использовать любое имя по своему выбору, но мы переименовали наш файл в `vsearch4web.py`. Вот код, который сейчас находится в этом файле.

Этот код сейчас находится в файле с именем «vsearch4web.py».

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

Чтобы отобразить HTML-форму с помощью шаблона `entry.html`, нужно внести несколько изменений в код выше.

1. **Импортировать функцию** `render_template`.
   Добавим `render_template` в список импорта, в строке `from flask` в начале программы.

2. ***Создать новый URL — в данном случае*** `/entry`.
   Каждый раз, когда вам понадобится добавить новый URL в Flask веб-приложение, добавьте также новую строку `@app.route`. Мы сделаем это перед строкой с вызовом `app.run()`.

3. ***Создать функцию, которая вернет сконструированную разметку HTML.***
   Со строкой `@app.route` можно связать функцию, которая выполнит всю фактическую работу (и сделает веб-приложение более полезным для пользователей). В этой функции мы вызовем `render_template` (и получим результат), передав ей имя файла шаблона (`entry.html` в этом случае), а также значения всех аргументов, необходимых шаблону (в этом случае мы должны передать значение для `the_title`).

Давайте внесем эти изменения в имеющийся код.

Отображение HTML-формы веб-приложения

Итак, внесем все три изменения, описанные в конце предыдущей страницы. Следуйте за нами и внесите такие же изменения в свой код:

1. **Импортировать функцию** `render_template`.

```
from flask import Flask, render_template
```

Добавьте «render_template» в список технологий, импортируемых из модуля «flask».

2. **Создать новый URL — в данном случае** `/entry`.

```
@app.route('/entry')
```

После функции «do_search», но перед строкой «app.run()» вставьте эту строку, чтобы добавить новый URL в веб-приложение.

3. **Создать функцию, которая вернет сконструированную разметку HTML.**

```
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')
```

Укажем имя шаблона для отображения.

Добавим эту функцию прямо под новой строкой «@app.route».

Передадим значение аргумента «the_title».

С изменениями, которые мы выделили фоном, код нашего веб-приложения должен выглядеть так.

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run()
```

Оставим остальную часть кода без изменений.

Подготовка к запуску кода с шаблонами

Очень заманчиво открыть командную строку и запустить последнюю версию кода. Однако по нескольким причинам прямо сейчас он не будет работать.

Если вы этого еще не сделали, скачайте шаблоны и CSS отсюда *http://python.itcarlow.ie/ed2/*.

Во-первых, базовый шаблон ссылается на таблицу стилей `hf.css`, которая должна находиться в папке `static` (внутри папки с вашим кодом). Это видно в следующем фрагменте базового шаблона.

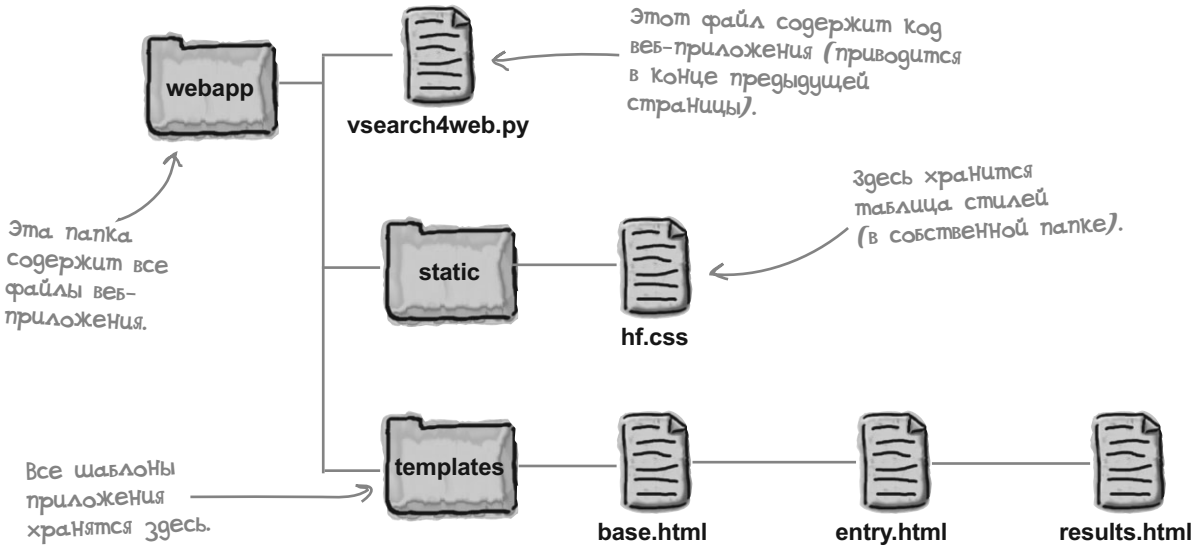
```
        ...
    <title>{{ the_title }}</title>
    <link rel="stylesheet" href="static/hf.css" />
</head>
        ...
```

Файл «hf.css» должен существовать (в папке «static»).

Вы можете взять копию CSS-файла с веб-сайта поддержки этой книги (смотрите URL на примечании сбоку). Не забудьте поместить загруженную таблицу стилей в папку `static`.

Во-вторых, Flask требует, чтобы шаблоны хранились в папке `templates`, которая — как и `static` — должна находиться внутри папки с вашим кодом. В пакете загружаемых примеров для этой главы содержатся все три шаблона... так что вы можете избежать ввода всего этого кода HTML!

С учетом того, что файл с кодом нашего веб-приложения находится в папке `webapp`, ниже показана структура каталогов и файлов, которая должна иметься у вас перед попыткой запустить наиболее свежую версию `vsearch4web.py`.



Мы готовы к тестовому запуску

Если у вас все готово — таблица стилей и шаблоны загружены, и код обновлен, — можно переходить к следующему циклу разработки веб-приложения.

Предыдущая версия все еще продолжает работать в командной строке.

Вернитесь в это окно и нажмите одновременно клавиши *Ctrl* и *C*, чтобы остановить веб-приложение. Затем нажмите клавишу со *стрелкой вверх*, чтобы ввернуть последнюю команду, исправьте имя запускаемого файла и нажмите клавишу *Enter*. Должна запуститься новая версия вашего кода и вывести привычное сообщение с информацией о состоянии.

```
        ...
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 21:51:38] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:51:48] "GET /search4 HTTP/1.1" 200 -
^C
$ python3 vsearch4web.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Остановим веб-приложение снова...

Запустим новую версию (в файле «vsearch4web.py»).

Новая версия запущена и ожидает запросов для обслуживания.

Эта новая версия все еще поддерживает URL */* и */search4*, соответственно, обратившись к ним из браузера, вы получите те же ответы, что были показаны выше в этой главе. Однако если ввести следующий URL:

http://127.0.0.1:5000/entry

браузер должен отобразить HTML-форму (показана в верхней части следующей страницы). В командной строке должны появиться два дополнительных сообщения: одно соответствует запросу с URL */entry*, а другое — запросу вашего браузера на получение таблицы стилей `hf.css`.

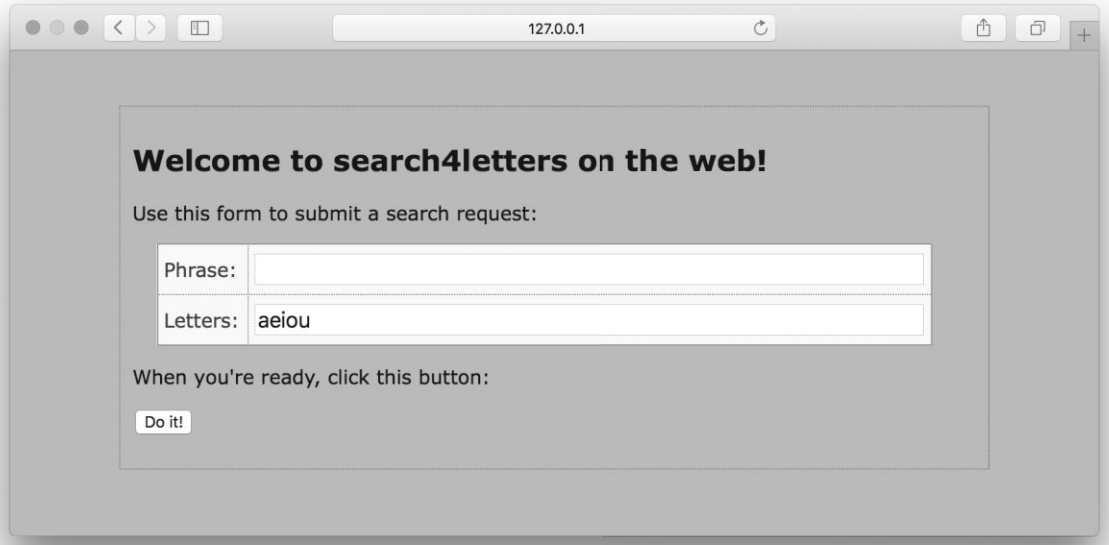
```
        ...
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /static/hf.css HTTP/1.1" 304 -
```

Вы запросили HTML-форму...

...дополнительно ваш браузер запросил таблицу стилей.

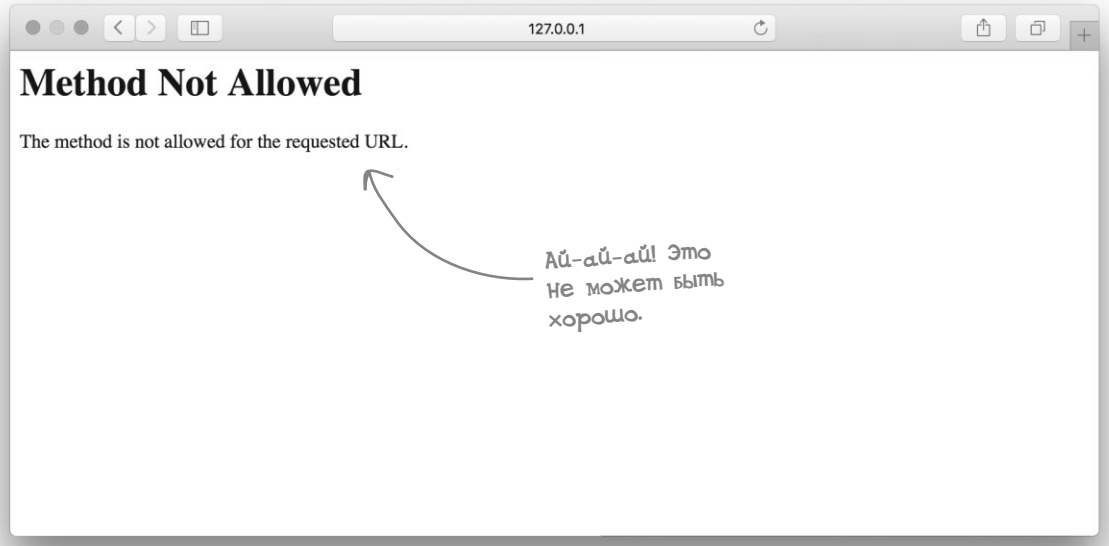
Пробная поездка

Вот что появляется на экране после ввода URL `http://127.0.0.1:5000/entry` в браузере:



Выглядит хорошо.

Мы не собираемся выиграть какую-нибудь награду в области веб-дизайна этой страничкой, но она выглядит нормально и напоминает рисунок на салфетке. К сожалению, после ввода фразы, букв (необязательно) в поле *Letters* и щелчка на кнопке *Do it!* выводится страничка с ошибкой:



Неприятно, правда? Давайте посмотрим, что произошло.

Коды состояния HTTP

Когда в веб-приложении что-то идет не так, веб-сервер отвечает кодом состояния HTTP (который отсылается вашему браузеру). HTTP — это коммуникационный протокол, позволяющий веб-браузерам и серверам общаться между собой. Это значит, что коды состояния хорошо известны (см. раздел *«Для умников»* справа). Фактически каждый *веб-запрос* генерирует *ответ* с кодом состояния HTTP.

Чтобы увидеть, какие коды состояния посылаются вашему браузеру, посмотрите на сообщения в командной строке. Вот что мы там видим:

```
    ...
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /static/hf.css HTTP/1.1" 304 -
127.0.0.1 - - [23/Nov/2015 21:56:54] "POST /search4 HTTP/1.1" 405 -
```

Ой-ой. Что-то пошло не так, и сервер сгенерировал код клиентской ошибки.=

Код 405 сообщает, что клиент (ваш браузер) послал запрос, используя HTTP-метод, который не поддерживается этим сервером. Существует несколько HTTP-методов, но для наших целей вам достаточно знать только о двух из них: *GET* и *POST*.

1. **Метод GET**
   Браузеры обычно используют этот метод для запроса ресурса с веб-сервера, и он, безусловно, является наиболее часто используемым. (Мы сказали «обычно», потому что метод *GET* можно также использовать для *отправки* данных из браузера на сервер, что довольно странно, но мы не будем заострять внимание на этой возможности.) Все URL из нашего веб-приложения сейчас поддерживают метод GET, который в Flask является HTTP-методом по умолчанию.

2. **Метод POST**
   Этот метод позволяет веб-браузеру посылать данные на сервер через HTTP, и он тесно связан с HTML-тегом `<form>`. Вы можете потребовать от своего Flask веб-приложения принимать данные, отправленные браузером, добавив еще один аргумент в строку `@app.route`.

Давайте откорректируем строку `@app.route`, определяющую URL */search4* в нашем веб-приложении, чтобы оно принимало отправленные данные. Для этого вернитесь в редактор и отредактируйте код в файле `vsearch4web.py` еще раз.

Для умников

Это краткое описание различных кодов состояния HTTP, которые может передавать веб-сервер (например, ваше веб-приложение) веб-клиенту (например, вашему веб-браузеру).

Есть пять основных категорий кодов состояния: 100-е, 200-е, 300-е, 400-е и 500-е.

Коды из диапазона **100–199** — это **информационные** сообщения: они говорят, что все в порядке, и сервер сообщает детали, относящиеся к клиентскому запросу.

Коды из диапазона **200–299** — это сообщения об **успехе**: сервер получил, понял и обработал запрос. Все в порядке.

Коды из диапазона **300–399** — это сообщения **перенаправления**: сервер информирует клиента, что запрос может быть обработан в каком-то другом месте.

Коды из диапазона **400–499** — это сообщения об **ошибках на стороне клиента**: сервер не смог понять и обработать запрос. Как правило, виноват в этом клиент.

Коды из диапазона **500–599** — это сообщения об **ошибках на стороне сервера**: сервер получил запрос, но в процессе обработки на сервере возникла ошибка. Как правило, виноват в этом сервер.

Более детальную информацию смотрите по адресу: *https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP*

Обработка отправленных данных

Помимо первого аргумента с URL, декоратор `@app.route` принимает и другие необязательные аргументы.

Один из них — аргумент `methods` со списком HTTP-методов, которые поддерживаются этим URL. По умолчанию Flask поддерживает GET для всех URL. Однако поведение по умолчанию изменяется, если аргумент `methods` содержит список поддерживаемых HTTP-методов. Вот как строка `@app.route` выглядит сейчас.

```
@app.route('/search4')
```

Мы не определили поддерживаемый HTTP-метод, поэтому Flask по умолчанию использует GET.

Чтобы URL */search4* поддерживал метод POST, добавьте в декоратор аргумент `methods` и присвойте ему список HTTP-методов, которые должны поддерживаться этим URL. Строка кода, приведенная ниже, сообщает, что URL */search4* теперь поддерживает только метод POST (то есть запросы GET больше не поддерживаются).

```
@app.route('/search4', methods=['POST'])
```

URL «/search4» поддерживает теперь только метод POST.

Этого маленького изменения достаточно, чтобы избавиться от сообщения «Method Not Allowed» («Метод не поддерживается»), потому что метод POST, используемый HTML-формой, соответствует методу POST в строке `@app.route`.

Этот фрагмент HTML взят из «entry.html...

```
...
<form method='POST' action='/search4'>
<table>
...
```

...а эта строка кода на Python взята из файла «vsearch4web.py».

```
...
@app.route('/search4', methods=['POST'])
def do_search() -> str:
...
```

Обратите внимание: в языке HTML используется существительное «method» (метод) в единственном числе, в то время как Flask использует существительное «methods» (методы) во множественном числе.

Это не Глупые вопросы

В: А если я хочу, чтобы мой URL поддерживал оба метода — не только GET, но и POST? Это возможно?

О: Да, для этого достаточно добавить имя нужного HTTP-метода в список, присваиваемый аргументу `methods`. Например, чтобы добавить поддержку GET для URL */search4*, надо лишь изменить строку кода `@app.route`, чтобы она выглядела так: `@app.route('/search4', methods=['GET', 'POST'])`. За подробностями обращайтесь к документации с описанием Flask, доступной по адресу: *http://flask.pocoo.org.*

Оптимизация цикла редактирование/остановка/запуск/проверка

Сейчас сохранение измененного кода является веской причиной для остановки веб-приложения в командной строке и последующего перезапуска для проверки нового кода. Этот цикл редактирование/остановка/запуск/проверка работает, но спустя время становится утомительным (особенно когда выполняется длинная серия мелких).

Чтобы повысить эффективность этого процесса, Flask позволяет запустить веб-приложение в *режиме отладки*, в котором, помимо прочего, автоматически перезапускает веб-приложение, когда замечает, что код изменился (обычно в результате внесения и сохранения изменений). Это стоит попробовать, так что давайте переключимся в режим отладки, изменив последнюю строку в файле `vsearch4web.py`, как показано ниже.

```
app.run(debug=True)
```

Переключение в режим отладки.

Код вашей программы должен сейчас выглядеть вот так.

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Теперь мы готовы его опробовать. Остановите веб-приложение (в последний раз), нажав *Ctrl-C*, и вновь запустите его в командной строке, нажав клавиши *со стрелкой вверх* и *Enter*.

Прежде чем показать обычное сообщение «`Running on http://127...`», Flask выведет три новых строки, сообщающие, что активирован режим отладки. Вот что он вывел на компьютере.

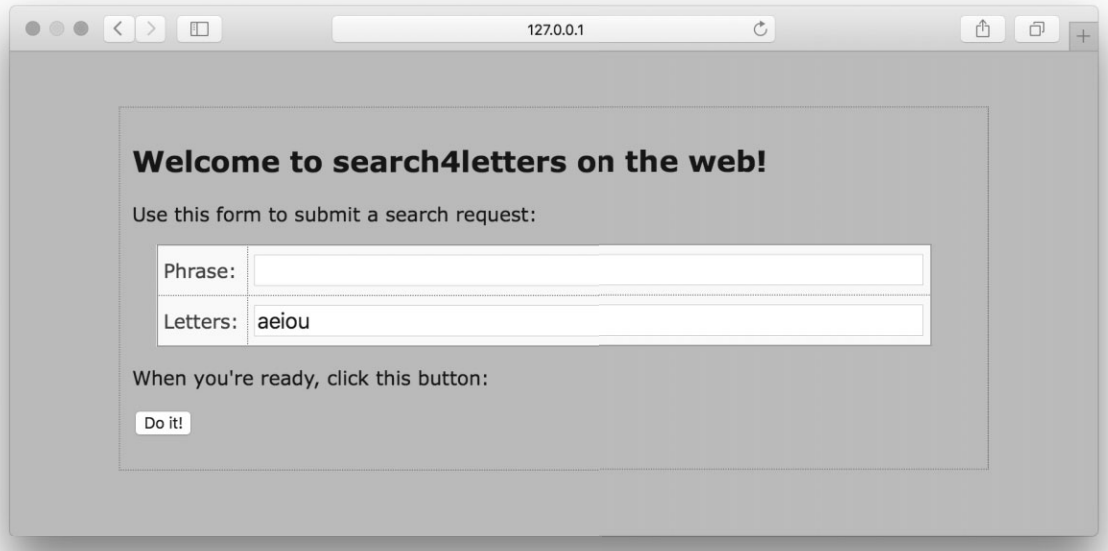
```
$ python3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

Так Flask сообщает, что веб-приложение будет автоматически перезапускаться после изменения кода. Кроме того, не беспокойтесь, если пин-код вашего отладчика отличается от нашего (это нормально). Мы этот пин-код использовать не будем.

После повторного запуска опробуем веб-приложение еще раз и посмотрим, что изменилось.

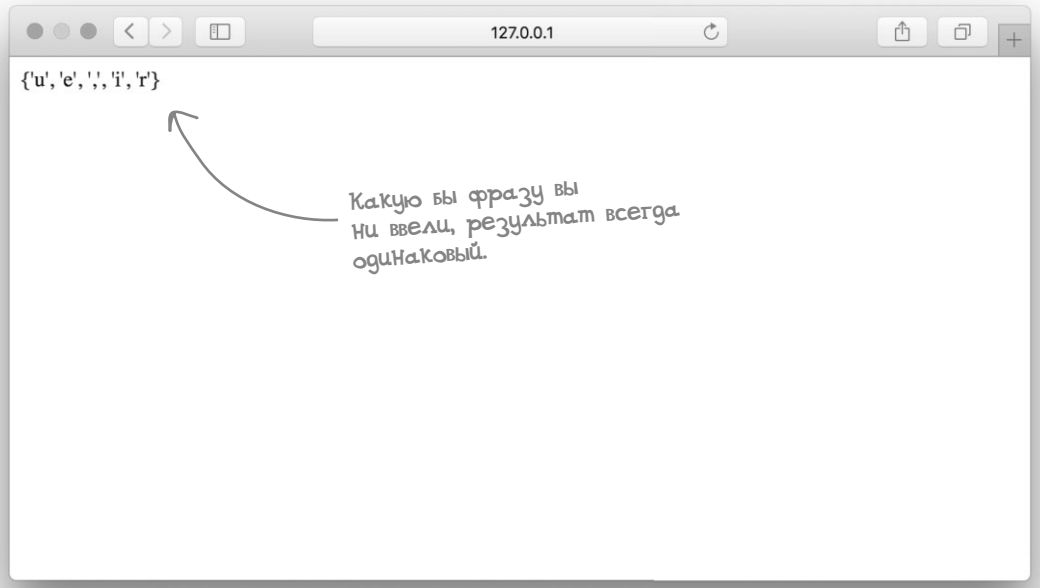
Пробная поездка

Вернемся в форму ввода данных, набрав ***`http://127.0.0.1:5000/entry`*** в браузере:



Все еще выглядит хорошо.

Ошибка «Method Not Allowed» пропала, но приложение все еще работает не так, как надо. Вы можете ввести любую фразу в эту форму и затем щелкнуть на кнопке *Do it!* — никакие ошибки не появятся. Но, проделав это несколько раз, вы заметите, что всегда возвращается один и тот же результат (какие бы фразы и буквы вы ни использовали). Нужно выяснить, в чем причина.



Доступ к данным HTML-формы с помощью Flask

Веб-приложение больше не ломается с ошибкой «Method Not Allowed». Но оно всегда возвращает один и тот же набор символов: *u, e, запятая, i* и *r*. Если заглянуть в код, который выполняется, когда на URL */search4* приходят данные, можно увидеть, почему так происходит: значения `phrase` и `letters` *жестко заданы* в коде функции.

```
                ...
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))
                ...
```

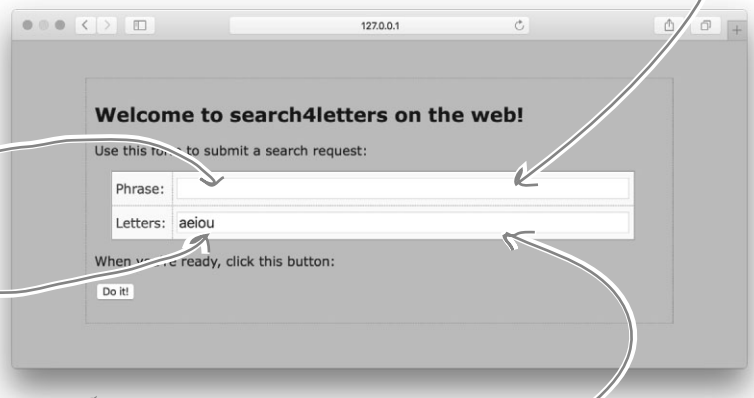
Не важно, что вы введете в HTML-форму: наш код всегда обрабатывает жестко заданные значения.

Наша HTML-форма посылает данные веб-серверу, но чтобы что-то сделать с этими данными, нужно изменить код веб-приложения, организовав прием данных перед выполнением любых операций с ними.

В состав Flask входит встроенный объект `request`, открывающий доступ к переданным данным. Объект `request` содержит атрибут-словарь `form` с данными HTML-формы, отправленной браузером. Подобно другим словарям в Python, `form` поддерживает ту же нотацию с квадратными скобками, которую вы впервые увидели в главе 3. Чтобы прочитать данные формы, поместим имя элемента ввода в форме внутрь квадратных скобок.

Данные из этого элемента ввода доступны в веб-приложении как «request.form['phrase']».

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```



HTML-шаблон (в файле «entry.html»).

Форма в Нашем веб-браузере.

Данные из этого элемента ввода доступны в веб-приложении как «request.form['letters']».

Использование данных запроса в веб-приложении

Чтобы воспользоваться объектом `request`, импортируйте его в строке `from flask` в начале программы. После этого вы сможете получить доступ к данным из `request.form`, когда будет нужно. Наша цель — заменить жестко заданные значения в функции `do_search` данными из формы. Благодаря этому каждый раз, когда в HTML-форму будут вводиться разные значения для `phrase` и `letters`, веб-приложение будет возвращать соответствующие им разные результаты.

Давайте внесем эти изменения. Начнем с добавления объекта `request` в список импорта из Flask. Для этого измените первую строку в файле `vsearch4web.py`, чтобы она выглядела так.

```
from flask import Flask, render_template, request
```

Добавили «request» в список импорта.

На предыдущей странице мы узнали, что к полю `phrase` HTML-формы можно обратиться как `request.form['phrase']`, а данные в поле `letters` доступны как `request.form['letters']`. Давайте откорректируем функцию `do_search` и используем эти значения (удалив жестко заданные строки).

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

Создаем две новые переменные...

...и присваиваем созданным переменным данные из HTML-формы...

...затем используем эти переменные в вызове «search4letters».

Автоматический перезапуск

Теперь... прежде чем вы сделаете что-то еще (после внесения предложенных изменений), сохраните файл `vsearch4web.py`, затем переключитесь в командную строку и посмотрите, какие сообщения вывело веб-приложение. Вот что увидели мы (вы должны увидеть нечто подобное).

```
$ python3 vsearch4web.py
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:17:58] "POST /search4 HTTP/1.1" 200 -
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

Отладчик Flask обнаружил, что код изменился и перезапустил веб-приложение. Очень удобно, правда?

Не паникуйте, если вы увидели что-то другое. Автоматический перезапуск происходит, только если изменения в коде являются корректными. Если код содержит ошибки, веб-приложение завершится и вернет управление командной строке. Чтобы возобновить работу, исправьте ошибки и запустите веб-приложение вручную (нажатием клавиш со *стрелкой вверх* и *Enter*).

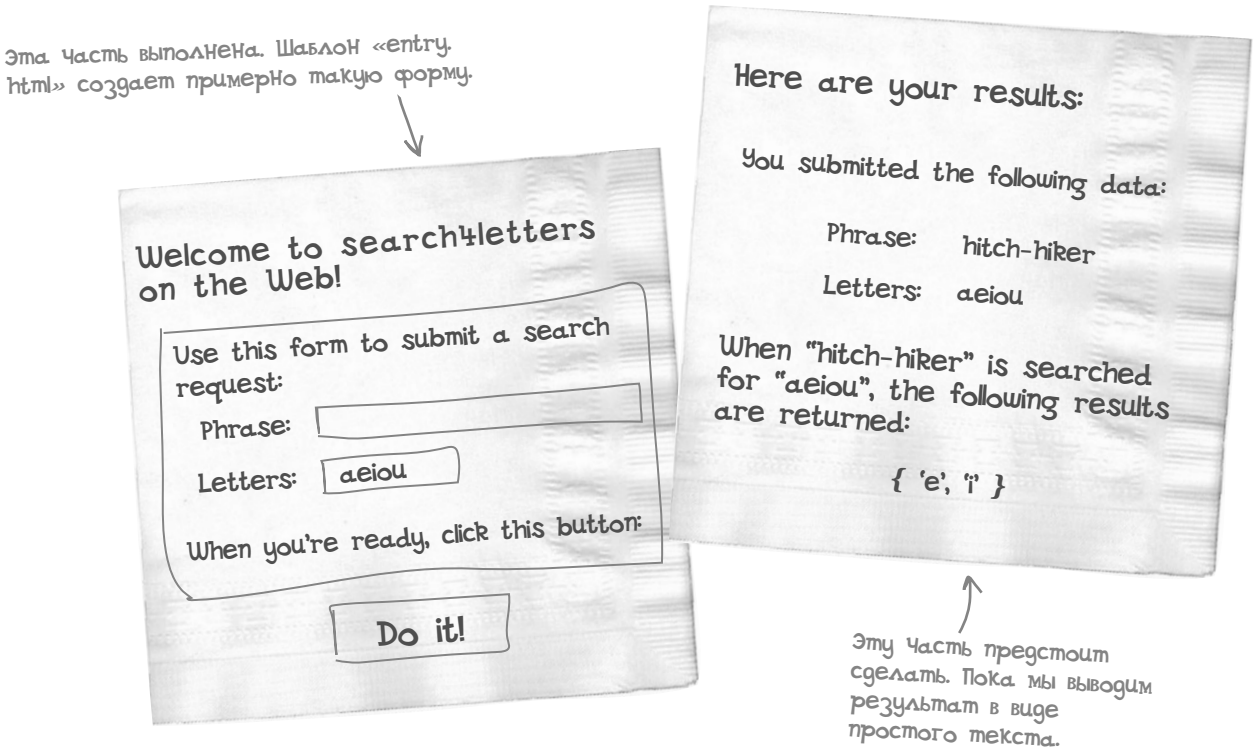
Пробная поездка

Теперь, когда наше веб-приложение принимает (и обрабатывает) данные из HTML-формы, мы можем передавать ему различные фразы и буквы, и оно должно работать правильно.

Выводим результат в виде HTML

На данный момент наше веб-приложение справляется со своими функциями: любой веб-браузер может передать комбинацию фраза/буквы, а веб-приложение вызовет `search4letters` от нашего имени и вернет результат. Однако этот результат на самом деле не является HTML-страничкой — это простые текстовые данные, возвращаемые ожидающему браузеру (который отображает их на экране).

Вспомним, что мы рисовали на салфетке ранее в этой главе. Вот что мы надеялись получить.



Изучая технологию шаблонов Jinja2, мы представили два HTML-шаблона. Первый, `entry.html`, предназначен для создания формы. Второй, `results.html`, — для отображения результатов. Давайте используем его и поместим наши текстовые данные в разметку HTML.

Это не Глупые вопросы

В: Можно ли использовать Jinja2 для включения текстовых данных в шаблон другого формата, отличного от HTML?

О: Да. Jinja2 — это механизм обработки текстовых шаблонов, который можно использовать для разных целей. Однако чаще всего он применяется в разработке веб-проектов (так как входит в состав Flask), но ничто не мешает использовать его с другими текстовыми данными, если это действительно нужно.

Вычисление нужных нам данных

Давайте напомним себе содержимое шаблона `results.html`, как он был представлен в этой главе. Разметка для Jinja2 выделена.

Вот «results.html».

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}", the following
results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

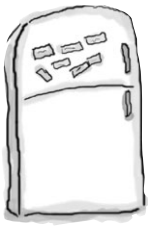
Выделенные имена в двойных фигурных скобках — это переменные Jinja2, которые принимают свои значения из соответствующих переменных в коде на Python. Вот эти переменные: `the_title`, `the_phrase`, `the_letters` и `the_results`. Взгляните еще раз на функцию `do_search` (ниже), в которую мы собираемся добавить отображение HTML-шаблона, приведенного выше. Функция уже содержит две из четырех переменных, необходимых для отображения шаблона (для простоты мы использовали в коде Python такие же имена переменных, как в шаблоне Jinja2).

Два из четырех нужных нам значений.

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

Чтобы определить остальные два аргумента шаблона (`the_title` и `the_results`), мы должны создать в функции соответствующие переменные и присвоить им значения.

Мы можем присвоить переменной `the_title` строку «`Here are your results:`» («`Ваши результаты:`»), переменной `the_results` — результат вызова `search4letters`, а затем передать в шаблон `results.html` все четыре переменные как аргументы.



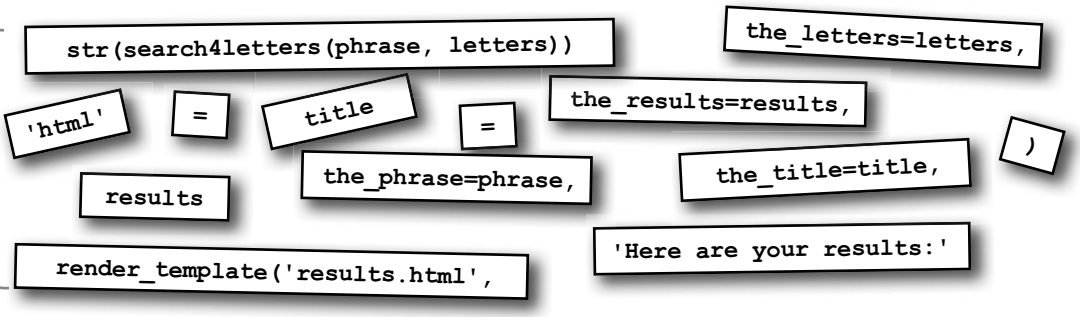
Шаблонные магнитики

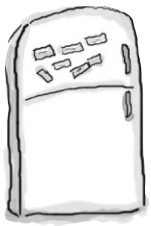
Авторы книги собрались вместе и, основываясь на требованиях к обновленной функции `do_search`, изложенных в конце предыдущей страницы, написали требуемый код. Следуя истинному стилю *Head First*, они сделали это с помощью магнитиков с кодом… и холодильника (лучше не спрашивайте). Во время празднования они настолько разошлись, что *редактор серии* врезался в холодильник (он пел *песню про пиво*). И вот — сейчас магнитики валяются на полу. Ваша задача — поместить их на свои места.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> ..........:
    phrase = request.form['phrase']
    letters = request.form['letters']
    ..........
    ..........
    return ..........
                          ..........
                          ..........
                          ..........
                          ..........

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Верните магнитики с кодом на места, отмеченные пунктирной линией.

Это магнитики, с которыми вам предстоит работать.





Решение шаблонных магнитиков

Заметив, что в будущем надо следить за потреблением пива редактором серии, вы приступили к задаче расстановки магнитиков, чтобы восстановить обновленную функцию `do_search`. Ваша задача — поместить магнитики на свои места.

Вот к чему пришли мы, когда решили задачу.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html' :
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_phrase=phrase,
                           the_letters=letters,
                           the_title=title,
                           the_results=results,
                           )

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Измените аннотацию, чтобы сообщить, что эта функция теперь возвращает HTML, а не обычную текстовую строку (как в предыдущей версии кода).

Создайте переменную Python и назовите ее «title»...

...присвойте «title» строку.

Создайте другую переменную с именем «results»...

...присвойте «results» результат вызова «search4letters».

Отобразите шаблон «results.html». Помните: этот шаблон ожидает значения четырех аргументов.

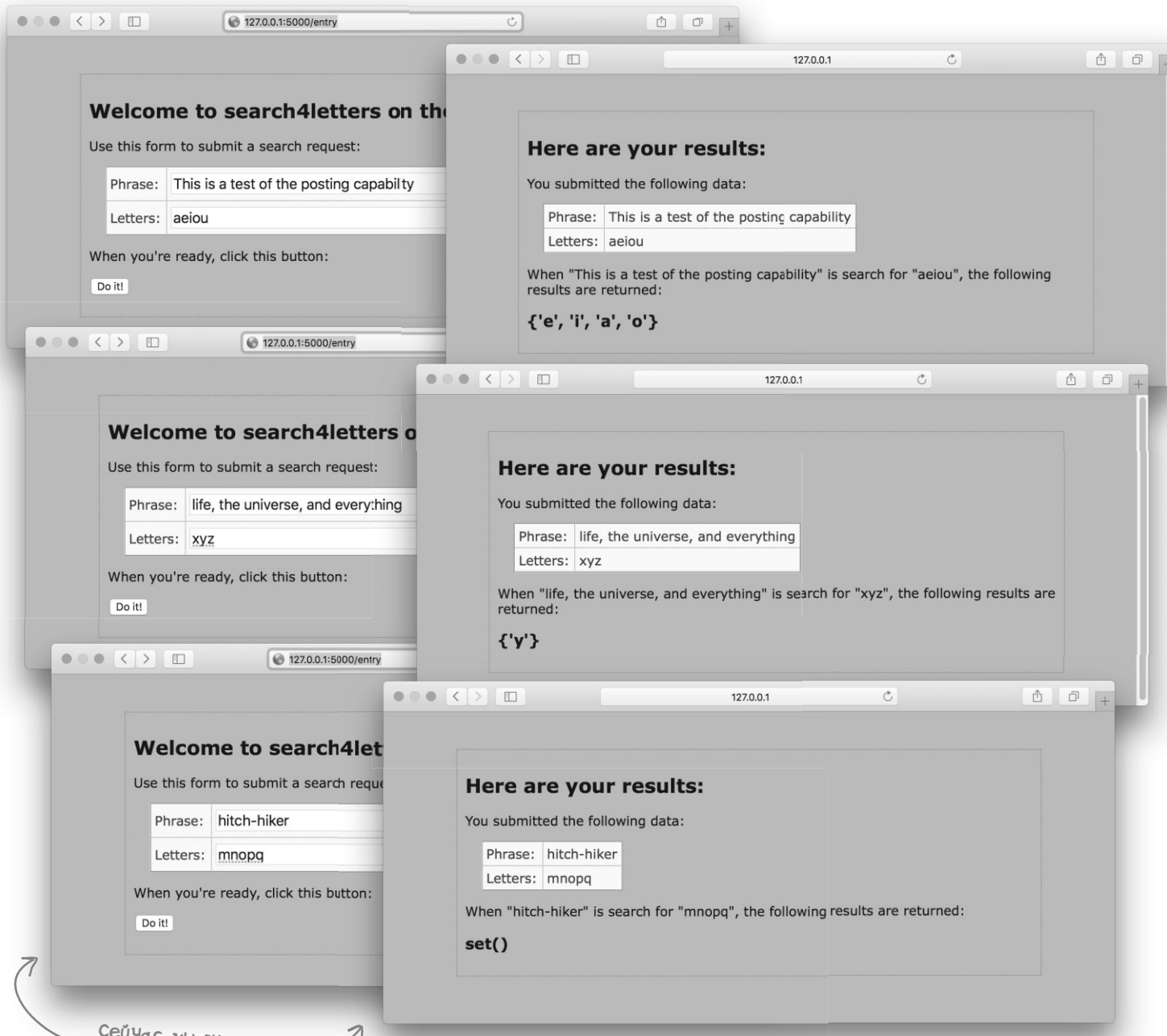
Свяжите каждую переменную с соответствующим аргументом. Таким характерным для Jinja2 способом данные из программного кода передаются в шаблон.

Не забудьте закрыть круглые скобки в конце вызова функции.

Теперь, когда все магнитики с кодом вернулись на места, внесите эти изменения в свою копию `vsearch4web.py`. Убедитесь, что сохранили файл, чтобы Flask автоматически перезагрузил веб-приложение. Теперь мы готовы к другой проверке.

Пробная поездка

Давайте проверим новую версию нашего веб-приложения, используя те же примеры, что были ранее в главе. Заметьте, что Flask перезапустил веб-приложение в тот момент, когда вы сохранили свой код.



Сейчас мы видим, что ввод и вывод выглядят хорошо.

Последние штрихи

Еще раз взглянем на код в `vsearch4web.py`. Надеемся, что сейчас он вам понятен. Один небольшой синтаксический элемент, который часто путает программистов, переходящих к Python, — это включение завершающей запятой в вызов `render_template`. Большинство программистов чувствуют, что такое недопустимо и должно вызвать синтаксическую ошибку. И хотя такое выглядит странно (поначалу), Python допускает это — хотя и не требует, — так что вы можете безопасно продолжать и не беспокоиться об этом.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Дополнительная запятая выглядит несколько странно, но при этом совершенно уместна (хотя ее можно опустить) и согласуется с синтаксисом Python.

Эта версия веб-приложения поддерживает три URL: */*, */search4* и */entry*. Некоторые из них появились в первой версии веб-приложения, созданного в начале главы. Сейчас URL */* показывает дружественное, хотя и бесполезное сообщение «Hello world from Flask!».

Мы могли бы удалить этот URL и ассоциированную с ним функцию `hello` из кода (поскольку мы в них больше не нуждаемся), но тогда мы будем получать ошибку 404 «Not Found» в любом веб-браузере, обратившемся к нашему веб-приложению по адресу URL */*, который является адресом URL по умолчанию для большинства веб-приложений и веб-сайтов. Для устранения этого досадного сообщения об ошибке давайте попросим Flask переадресовывать любые запросы с URL */* на URL */entry*. Для этого мы изменим функцию `hello` так, чтобы она возвращала команду перенаправления любому веб-браузеру, обратившемуся к URL */*, фактически подставляя URL */entry* в любые запросы к адресу */*.

Перенаправление для устранения нежелательных ошибок

Чтобы воспользоваться возможностью перенаправления из фреймворка Flask, добавьте `redirect` в список импорта `from flask` (в начале программы), а затем измените код функции `hello`, как показано ниже.

Добавьте «redirect» в список импорта.

```
from flask import Flask, render_template, request, redirect
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> '302':
    return redirect('/entry')

    ...
```

Откорректируйте аннотацию, чтобы ясно определить, что будет возвращено этой функцией. Помните: коды состояния HTTP в диапазоне 300–399 — это перенаправления, и код 302 Flask отправит браузеру в случае вызова «redirect».

Остальной код без изменений.

Вызов функции «redirect» сообщает браузеру, что нужно запросить альтернативный URL (в нашем случае «/entry»).

Это небольшое изменение гарантирует, что пользователи нашего веб-приложения увидят HTML-форму при обращении к URL */entry* или */*.

Внесите изменения, сохраните код (это вызовет автоматический перезапуск) и попробуйте ввести в своем браузере каждый из этих URL. В обоих случаях должна появляться HTML-форма. Взгляните на сообщения, которые выводит веб-приложение в командной строке. Вы можете увидеть нечто похожее на это:

Запрос был выполнен на URL «/entry» и обработан незамедлительно. Обратите внимание на код состояния 200 (и вспомните, что коды 200–299 сообщают об успехе: сервер получил, понял и обработал клиентский запрос).

```
    ...
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [24/Nov/2015 16:54:13] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [24/Nov/2015 16:56:43] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [24/Nov/2015 16:56:44] "GET /entry HTTP/1.1" 200 -
```

Вы сохранили код, поэтому Flask перезапустил ваше веб-приложение.

Получив запрос на URL «/», веб-приложение отвечает перенаправлением, возвращая код 302, а затем веб-браузер посылает другой запрос на URL «/entry», который успешно обрабатывается веб-приложением (еще раз обратите внимание на код состояния 200).

Стратегически прием с перенаправлением работает, но он влечет напрасный расход системных ресурсов — один запрос на URL */* выливается в два запроса (кеширование на стороне клиента может помочь, но это все равно не оптимально). Если бы только Flask мог как-то связать с заданной функцией несколько URL, эффективно устраняя необходимость в перенаправлении вообще. Это было бы здорово, верно?

Функции могут обслуживать множество URL

Легко догадаться, куда мы клоним, не так ли?

Оказывается, Flask может ассоциировать с одной функцией несколько URL, что позволяет избавиться от перенаправлений, показанных на предыдущей странице. Когда с функцией связано несколько URL, Flask проверяет каждый из них и, найдя соответствие, вызывает функцию.

Использовать такую возможность Flask совсем не трудно. Для начала удалите `redirect` из списка импорта `from flask` в начале программы; мы более не нуждаемся в этой функции, поэтому не будем импортировать код, который не собираемся использовать. Далее, используя редактор, вырежьте строку кода `@app.route('/')` и вставьте ее перед строкой `@app.route('/entry')` ближе к концу файла. Наконец, удалите две строки кода, составляющие функцию `hello`, потому что веб-приложение более не нуждается в ней.

После этих изменений код программы должен выглядеть так.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Нам больше не надо импортировать функцию «redirect», поэтому мы удалили ее из списка импорта.

Функция «hello» также была удалена.

Функция «entry_page» теперь связана с двумя URL.

Сохранив этот код (что вызовет перезапуск), мы сможем проверить новую функциональность. Если ввести в браузере URL /, появится HTML-форма. Сообщения о состоянии веб-приложения подтверждают, что обработка / выполняется в одном запросе, а не в двух (как в предыдущем случае).

```
    ...
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [24/Nov/2015 16:59:10] "GET / HTTP/1.1" 200 -
```

Как всегда, новая версия веб-приложения автоматически перезапускается.

Один запрос, один ответ. Так-то лучше ☺

Повторим, что мы узнали

На предыдущих 40 страницах мы занимались созданием небольшого веб-приложения, которое открывает доступ к функции `search4letters` из Всемирной паутины (при помощи простого двухстраничного сайта). Сейчас веб-приложение выполняется локально на вашем компьютере. Скоро мы расскажем, как развернуть веб-приложение в облаке, а сейчас повторим то, что нам известно.

КОНТРОЛЬНЫЙ СПИСОК

- Вы узнали о Python Package Index (PyPI) — централизованном хранилище сторонних модулей для Python. При наличии подключения к Интернету вы можете автоматически устанавливать пакеты из PyPI, используя `pip`.
- Использовали `pip` для установки микро-веб-фреймворка **Flask**, на основе которого затем построили веб-приложения.
- Значение `__name__` (поддерживаемое интерпретатором) определяет текущее активное пространство имен (подробнее об этом далее).
- Символ `@` перед именем функции определяет ее как **декоратор**. Декораторы позволяют изменить поведение существующей функции, не внося изменения в ее код. В веб-приложении вы использовали декоратор `@app.route` из Flask для связывания URL с функциями Python. Функцию можно декорировать несколько раз (как показано в примере с функцией `do_search`).
- Вы научились использовать механизм текстовых шаблонов **Jinja2** для отображения HTML-страниц внутри веб-приложения.

И это все, что есть в этой главе?

Если вы считаете, что в этой главе не так уж много нового, то вы правы. Одной из целей этой главы было показать, как мало строк кода на Python нужно, чтобы создать что-то довольно полезное в веб, хотя и во многом благодаря фреймворку Flask. Использование технологии шаблонов также полезно и позволяет держать код на Python (логику) отдельно от HTML-страничек (пользовательского интерфейса) веб-приложения.

Расширить веб-приложение, чтобы оно делало больше, не составит большого труда. Фактически вы могли бы привлечь юного вундеркинда для создания HTML-страниц, а сами сосредоточиться на написании кода на Python, связывающем все вместе. По мере роста веб-приложения такое разделение обязанностей начинает окупаться. Вы занимаетесь кодом на Python (как программист проекта), а ваш вундеркинд концентрируется на разметке HTML (поскольку это его коронная область). Конечно, вы оба должны немного знать о шаблонах Jinja2, но это не очень-то и сложно?

Подготовка веб-приложения к развертыванию в облаке

Теперь, когда веб-приложение готово и работает на локальном компьютере согласно исходным требованиям, пора подумать о его развертывании для использования широкой аудиторией. Есть масса возможностей в виде большого количества различных предложений по оказанию услуг веб-хостинга с поддержкой Python. Одно из популярных предложений — услуга *PythonAnywhere* хостинга в облаке на AWS. Мы в *лаборатории Head First* выбираем ее.

Подобно большинству решений облачного хостинга, служба *PythonAnywhere* предпочитает автоматически управлять запуском веб-приложения. Для вас это означает, что *PythonAnywhere* берет на себя ответственность за вызов `app.run()` от вашего имени, то есть вам больше не надо вызывать `app.run()` в коде. Фактически, если вы попытаетесь выполнить эту строку кода, *PythonAnywhere* просто откажется запускать веб-приложение.

Самое простое решение этой проблемы — удалить последнюю строку кода из вашего файла *перед* развертыванием в облаке. Прием действительно работает, но это означает, что вам придется вернуть строку обратно, когда понадобится запустить веб-приложение локально. Если вы пишете и проверяете новый код, то должны делать это на локальном компьютере (не в *PythonAnywhere*), так как облако используется только для развертывания, а не для разработки. Кроме того, прием с удалением проблемной строки вынуждает поддерживать две версии одного веб-приложения — одну с этой строкой кода и другую без нее. В общем, это не самое удачное решение, которое к тому же усложняет управление исходным кодом с увеличением изменений.

Было бы неплохо, если бы существовал способ выборочного выполнения кода в зависимости от места запуска веб-приложения: на локальном компьютере или удаленно, под управлением *PythonAnywhere*...

Да, это отличное предложение.

Эта конкретная строка кода *используется* во многих программах на Python и проверяет, был ли запущен этот файл как самостоятельная программа. Чтобы понять, как она может пригодиться (и как ее использовать с *PythonAnywhere*), разберемся, что она делает и как работает.

Переменная __name__ под лупой

Чтобы понять суть программной конструкции, предложенной в конце предыдущей страницы, рассмотрим маленькую программу `dunder.py`, которая ее использует. Программа из трех строк начинается с вывода сообщения, содержащего название текущего активного пространства имен, которое хранится в переменной `__name__`. Затем инструкция `if` проверяет равенство значения `__name__` строке `'__main__'` и, если условие выполняется, выводит другое сообщение, подтверждающее значение `__name__` (то есть выполняется тело инструкции `if`):

Код программы «dunder.py» – все три строки.

```
print('We start off in:', __name__)
if __name__ == '__main__':
    print('And end up in:', __name__)
```

Выводит значение «__name__».

Выводит значение «__name__», если оно равно строке «__main__».

С помощью своего редактора (или IDLE) создайте файл `dunder.py`, запустите программу из командной строки и посмотрите, что случится. Если вы работаете в *Windows*, используйте команду:

```
C:\> py  -3  dunder.py
```

Если вы работаете в *Linux* или *Mac OS X*, используйте команду:

```
$ python3  dunder.py
```

Независимо от операционной системы, программа `dunder.py` – при *непосредственном* выполнении – выведет на экран:

```
We start off in: __main__
And end up in: __main__
```

Когда программа выполняется непосредственно, на экране появляется вывод от обоих вызовов «print».

Пока все хорошо.

Теперь посмотрим, что случится, если импортировать файл `dunder.py` (который *также* является модулем) в командной строке >>>. Мы здесь покажем вывод в *Linux/Mac OS X*. Чтобы сделать то же самое в *Windows*, замените `python3` (ниже) на `py -3`:

```
$ python3
Python 3.5.1 ...
Type "help", "copyright", "credits" or "license" for more information.
>>> import dunder
We start off in: dunder
```

Посмотрите на это: появилась только одна строка (а не две), потому что в переменную «__name__» было записано значение «dunder» (что соответствует имени импортируемого модуля).

Вот то немногое, что вы должны понять: если программа выполняется *непосредственно*, то выражение в инструкции `if` в `dunder.py` возвращает `True`, потому что активное пространство имен имеет имя `'__main__'`. Однако если программный код импортируется как модуль (как в примере с интерактивной оболочкой Python выше), то выражение в инструкции `if` всегда возвращает `False`, потому что переменная `__name__` содержит не строку `'__main__'`, а имя импортируемого модуля (в нашем случае `dunder`).

Используем переменную __name__

Теперь, когда вы знаете, как работает переменная `__name__`, используем ее для решения проблемы со службой *PythonAnywhere*, желающей самой выполнять `app.run()` от нашего имени.

Оказывается, что *PythonAnywhere* выполняет веб-приложение, импортируя файл с его кодом, как любой другой модуль. Если импортирование прошло успешно, *PythonAnywhere* вызывает `app.run()`. Это объясняет, почему вызов `app.run()` в конце нашей программы представляет проблему для *PythonAnywhere* — эта служба предполагает, что вызов `app.run()` не был сделан, и не может запустить веб-приложение, когда оно само вызывает `app.run()`.

Чтобы обойти проблему, обернем вызов `app.run()` инструкцией `if`, проверяющей переменную `__name__` (которая предотвратит вызов `app.run()`, когда веб-приложение импортируется как модуль).

Отредактируйте `vsearch4web.py` еще один, последний раз (по крайней мере в этой главе) и измените последнюю строку, как показано ниже.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Вызов «app.run()» теперь производится, только когда программа запускается непосредственно.

Это небольшое изменение позволяет продолжать запускать веб-приложение локально (где строка `app.run()` *будет* выполняться), а также разворачивать его в *PythonAnywhere* (где строка `app.run()` *не будет* выполняться). Независимо от того, где запущено веб-приложение, вы получили единую версию, работающую правильно везде.

Развертывание в PythonAnywhere (ну... почти)

Теперь осталось только фактически развернуть веб-приложение в облачном окружении *PythonAnywhere*.

Заметьте также, что для целей этой книги развертывание веб-приложения в облаке *не* является абсолютно необходимым. В следующей главе мы намерены расширить возможности `vsearch4web.py`, но вам и далее не потребуется развертывать приложение в *PythonAnywhere*. Вы можете спокойно продолжать редактировать/запускать/проверять веб-приложение локально, когда мы будем расширять его в следующей главе (и далее).

Однако если вы действительно хотите развернуть приложение в облаке, обращайтесь к *Приложению Б*, где приводится пошаговая инструкция по полному развертыванию в *PythonAnywhere*. Это несложно и займет не более 10 минут.

Займетесь ли вы сейчас развертыванием в облаке или нет, мы увидимся в следующей главе и ознакомимся с возможностями сохранения данных из программ на Python.

Код из главы 5

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

Это «hello_flask.py», первое веб-приложение, основанное на Flask (одном из микро-веб-фреймворков на Python).

Это «vsearch4web.py» — веб-приложение, открывающее доступ к нашей функции «search4letters» из Всемирной паутины. Кроме Flask, этот код использует механизм шаблонов Jinja2.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to... web!')

if __name__ == '__main__':
    app.run(debug=True)
```

Программа «dunder.py», помогающая понять очень удобный механизм проверки переменной «__name__».

```
print('We start off in:', __name__)
if __name__ == '__main__':
    print('And end up in:', __name__)
```

6 Хранение и обработка данных

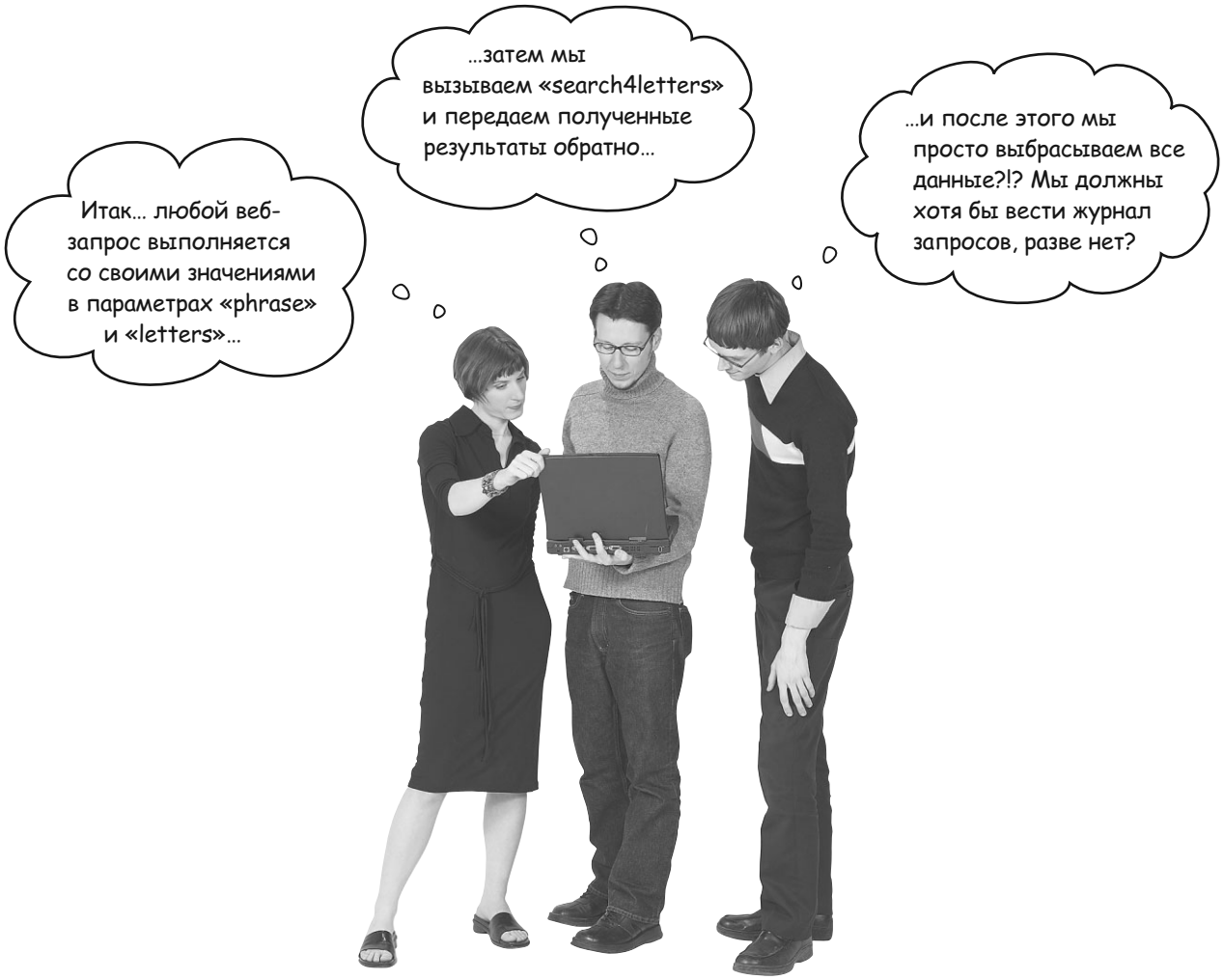
Где хранить данные

Рано или поздно появляется необходимость обеспечить надежное хранение данных.

И когда придет время **сохранить данные**, Python вам поможет. В этой главе вы узнаете о хранении и извлечении данных из *текстовых файлов*, которые — как механизм хранения — могут показаться слишком простыми, но тем не менее используются во многих проблемных областях. Кроме сохранения и извлечения данных из файлов, вы научитесь некоторым премудростям при работе с данными. «Серьезный материал» (хранение информации в базе данных) мы припасли для следующей главы, однако с файлами тоже придется потрудиться.

Работа с данными из веб-приложения

Теперь ваше веб-приложение (разработанное в главе 5) принимает входные данные от любого браузера (в форме параметров запроса `phrase` и `letters`), выполняет вызов `search4letters` и возвращает результаты браузеру. А потом теряет все свои данные.



Есть несколько вопросов, которые можно задать по поводу данных, используемых в веб-приложении. Например: *Сколько запросов было обработано? Какой список букв встречается наиболее часто? С каких IP-адресов приходили запросы? Какой браузер использовался чаще?*

Чтобы ответить на эти и другие вопросы, мы должны не выбрасывать данные веб-приложения, а сохранять их. Предложение выше не лишено смысла: давайте регистрировать каждый веб-запрос, а затем — когда появится механизм журналирования — мы сможем ответить на любой возникший вопрос.

Python позволяет открывать, обрабатывать и закрывать

Самый простой способ сохранить данные (независимо от языка программирования) — записать их в текстовый файл. В Python также есть встроенные инструменты, позволяющие *открыть, обработать* и *закрыть*. Используя эту стандартную методику, вы сможете **открыть** файл, **обработать** данные, хранящиеся в нем (прочитать, записать и/или добавить данные в конец файла), а по завершении **закрыть** файл (сохранив изменения).

Посмотрим, как использовать методику *открыть, обработать, закрыть* в языке Python, чтобы открыть файл, обработать, записав в конец несколько коротких строк, и закрыть его. Раз уж мы экспериментируем, опробуем наш код в интерактивной оболочке Python.

Вначале вызовем функцию `open`, чтобы открыть файл `todos.txt` в режиме *добавления в конец* (*append*), потому что мы планируем дописывать данные в конец файла. В случае успеха вызов `open` вернет объект (известный как *файловый поток*), служащий псевдонимом фактического файла. Объект присваивается переменной с именем `todos` (имя файла `todos` переводится как «список дел». — *Прим. науч. ред.*), хотя вы можете выбрать любое другое имя.

Для умников

Чтобы получить доступ к командной строке >>>:

- запустите IDLE;
- выполните команду `python3` в терминале *Linux* или *Mac OS X*; или
- выполните `py -3` в командной строке *Windows*.

Открыть файл... ...с этим именем...

```
>>> todos = open('todos.txt', 'a')
```

В случае успеха «open» вернет файловый поток, который мы присвоим этой переменной.

...в режиме «добавления в конец».

Переменная `todos` позволяет обращаться к файлу из программного кода (в других языках программирования такие переменные называются *дескрипторами файлов*). Теперь, когда файл открыт, запишем в него данные с помощью `print`. Обратите внимание: `print` принимает дополнительный аргумент (`file`), идентифицирующий файловый поток для записи. Нам нужно запомнить три дела (хотя список дел можно продолжать до бесконечности), поэтому вызовем `print` три раза.

Выведем сообщение... ...в файловый поток.

```
>>> print('Put out the trash.', file=todos)
>>> print('Feed the cat.', file=todos)
>>> print('Prepare tax return.', file=todos)
```

Поскольку нам больше нечего добавить в список дел (to-do), закроем файл, вызвав метод `close`, доступный для каждого файлового потока.

```
>>> todos.close()
```

Мы поработали, теперь приберем за собой, закрыв файловый поток.

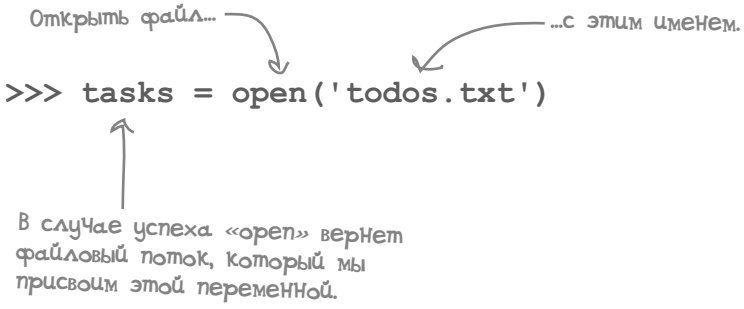
Если забыть вызвать `close`, то можно потерять данные. Поэтому важно помнить о необходимости вызова `close`.

Чтение данных из существующего файла

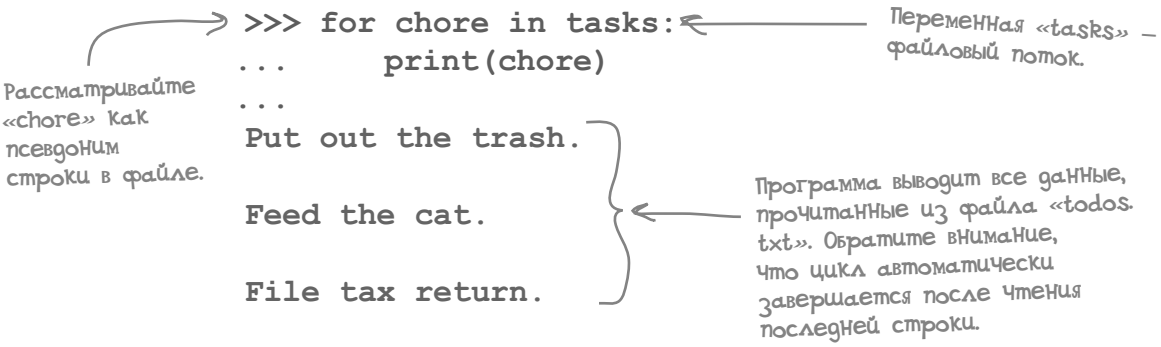
Теперь, когда мы добавили несколько строк в файл `todos.txt`, посмотрим, как в методике *открыть, обработать, закрыть* выглядит код, позволяющий прочитать данные из файла и вывести их на экран.

«Чтение» — режим по умолчанию для функции «open».

На этот раз мы откроем файл не для добавления в конец, а для чтения. Режим чтения используется функцией `open` **по умолчанию**, поэтому не нужно указывать какой-либо аргумент — достаточно имени файла. Мы не станем использовать переменную `todos` как псевдоним файла в этом коде, а будем ссылаться на открытый файл по имени `tasks` (как и прежде, вы можете выбрать для переменной другое имя).



Теперь используем `tasks` в цикле `for`, чтобы прочитать каждую строку из файла и присвоить ее переменной цикла (`chore`). В каждой итерации переменной `chore` будет присваиваться очередная строка из файла. Когда файловый поток используется с циклом `for`, интерпретатор Python берет на себя чтение очередной строки из файла в каждой итерации. Кроме того, он самостоятельно завершит цикл, достигнув конца файла.



Если вы только читаете из существующего файла, вызов `close` менее критичен, чем в случае записи данных. Но все же лучше закрыть файл, если он больше не нужен, поэтому вызовем метод `close`.

Это не Глупые вопросы

В: **А что там с дополнительными пустыми строчками при выводе? У нас в файле только три строки, а цикл `for` вывел шесть. Как это вышло?**

О: Да, вывод цикла `for` выглядит странно. Дело в том, что функция `print` добавляет символ перевода строки после всего, что выводит на экран, и *это ее поведение по умолчанию*. К тому же каждая строка в файле заканчивается символом перевода строки (и он читается как часть строки). Вот и получается, что фактически выводятся два символа перевода строки: один из файла, а другой добавляет функция `print`. Чтобы `print` не добавляла символ перевода строки, замените `print(chore)` на `print(chore, end='')`. В этом случае функция `print` не будет выводить второй символ перевода строки и на экране не будут появляться пустые строки.

В: **Какие еще режимы доступны для работы с данными в файлах?**

О: Есть еще несколько режимов, мы их опишем в блоке «Для умников» (хороший вопрос, кстати).

Для умников

Первый аргумент функции `open` — имя файла для обработки. А второй аргумент **можно опустить**. В нем можно передавать разные значения, определяющие **режим** открытия файла. В число режимов входят: «чтение», «запись» и «добавление в конец». Вот наиболее часто встречающиеся значения режимов, каждое из которых (кроме `r`) создает пустой файл, если файл с таким именем отсутствует:

`'r'` открыть файл для **чтения**. Это режим по умолчанию, поэтому аргумент с данным значением можно опустить. Если в вызове функции `open` второй аргумент отсутствует, то подразумевается `'r'`. Также подразумевается, что файл для чтения уже существует.

`'w'` открыть файл для **записи**. Если в файле уже есть данные, они удаляются.

`'a'` открыть файл для **добавления в конец**. Сохраняет содержимое файла, добавляя данные в конец файла (сравните это поведение с режимом `'w'`).

`'x'` открыть **новый файл** для записи. Вызов завершается неудачей, если файл уже существует (сравните с `'w'` и `'a'`).

По умолчанию файл открывается в **текстовом** режиме, когда подразумевается, что файл содержит строки с текстовыми данными (например, ASCII или UTF-8). Если вы работаете с нетекстовыми данными (например, с изображением или аудиофайлом MP3), нужно указать, что файл должен открываться в **двоичном** режиме, добавив символ `'b'` к любому из режимов, перечисленных выше (например, `'wb'` означает «режим записи двоичных данных»). Если вы добавите символ `'+'` во второй аргумент, файл будет открыт в режиме для чтения *и* записи (например, `'x+b'` означает «читать и записывать в новый двоичный файл»). Чтобы узнать больше о работе `open` (и других опциональных аргументах), обращайтесь к документации по Python.

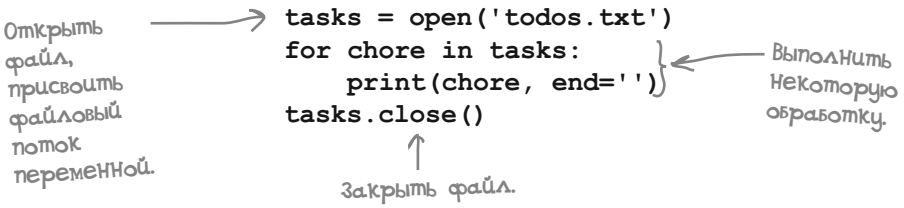
Я посмотрел проекты в репозитории GitHub. В большинстве из них при открытии файлов используется выражение «with». Что это значит?

Инструкция *with* удобнее.

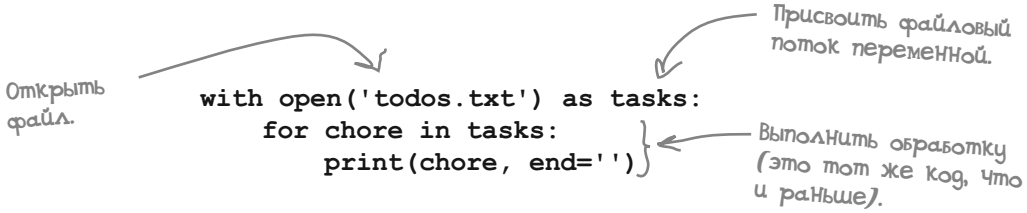
Использование функции `open` в паре с методом `close` (и дополнительными операциями между их вызовами) позволяет добиться желаемого, но большинство программистов на Python предпочитают вместо методики *открыть*, *обработать*, *закрыть* использовать инструкцию `with`. Давайте узнаем почему.

Лучше «with», чем открыть, обработать, закрыть

Прежде чем описать, почему инструкция `with` так популярна, посмотрим, как ею пользоваться. Вот код, который читает и выводит на экран содержимое файла `todos.txt`. Обратите внимание: мы настроили функцию `print`, чтобы она не выводила дополнительные пустые строки.



Теперь перепишем код с использованием инструкции `with`. Следующие три строки кода используют `with`, чтобы сделать *то же самое*, что делают четыре строки кода выше.



Заметили, что исчезло? Исчез вызов `close`. Инструкция `with` автоматически вызывает `close`, когда блок кода завершает выполнение.

Это намного полезнее, чем может показаться сначала, потому что многие программисты часто забывают вызвать `close`, закончив работу с файлом. Это не так важно, когда вы читаете файл, но если не вызвать `close` после записи в файл, то можно *потерять или испортить данные*. Избавляя от необходимости помнить о вызове `close`, инструкция `with` позволяет сконцентрироваться на обработке данных в открытом файле.

Python поддерживает методику «открыть, обработать, закрыть». Однако большинство программистов предпочитают использовать инструкцию with.

Инструкция «with» управляет контекстом

Инструкция `with` соответствует соглашению, реализованному в Python, которое называется **протокол управления контекстом**. Мы рассмотрим этот протокол позже. А сейчас запомните: при использовании `with` для обработки файлов можно смело забыть о вызове `close`. Инструкция `with` управляет контекстом, в котором выполняется блок кода, и если `with` и `open` используются вместе, интерпретатор уберет за вами, вызвав `close`, когда нужно.

Упражнение

Попробуем использовать все, что мы узнали о файлах. Ниже приведен код нашего веб-приложения в его текущем состоянии. Прочтите его еще раз, прежде чем мы скажем, что нужно делать:

Код «vsearch4web.py» из главы 5.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

if __name__ == '__main__':
    app.run(debug=True)
```

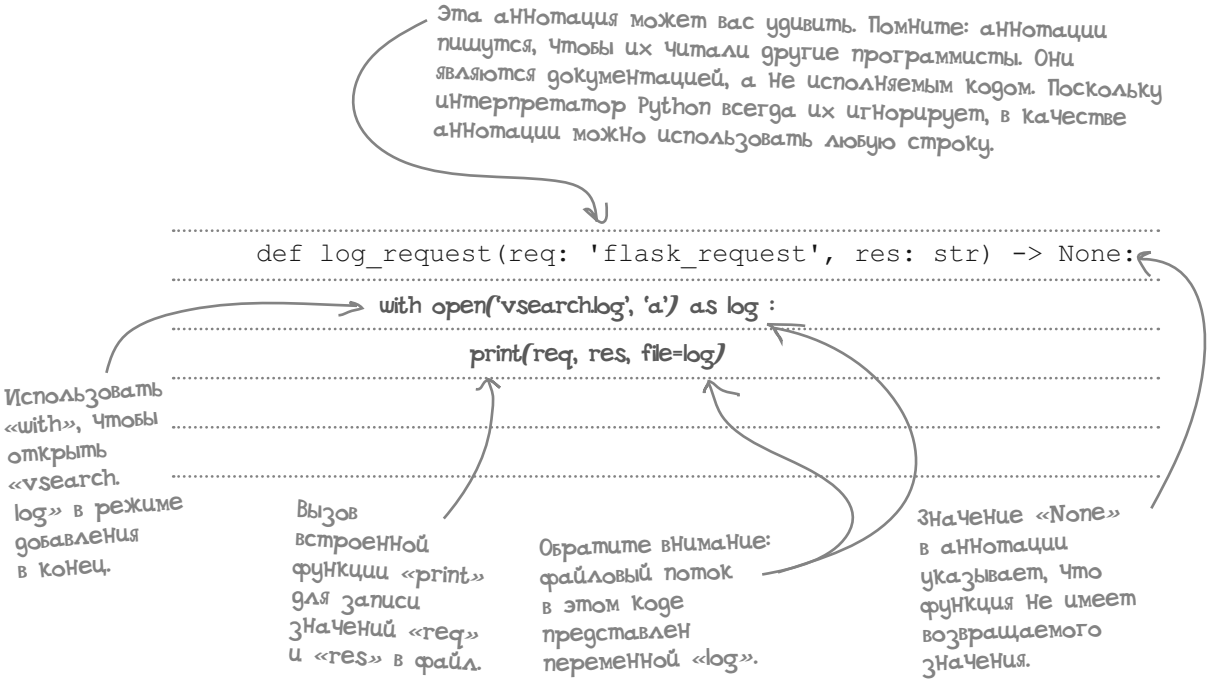
Ваша задача — написать новую функцию `log_request`, которая принимает два аргумента: `req` и `res`. В аргументе `req` передается текущий объект запроса, а в аргументе `res` — результат вызова `search4letters`. Функция `log_request` должна записать в конец файла `vsearch.log` значения `req` и `res` (в одну строку). Мы предоставим вам только строку `def` с заголовком функции. Пропущенный код нужно добавить (подсказка: используйте `with`).

Напишите здесь тело функции.

```
def log_request(req: 'flask_request', res: str) -> None:
```

Упражнение
Решение

Ваша задача — написать новую функцию `log_request`, которая принимает два аргумента: `req` и `res`. В аргументе `req` передается текущий объект запроса, а в аргументе `res` — результат вызова `search4letters`. Функция `log_request` должна записать в конец файла `vsearch.log` значения `req` и `res` (в одну строку). Мы положили начало этой работе, а вам надо было вписать недостающий код.



Вызов функции журналирования

Теперь, когда у нас появилась функция `log_request`, осталось выяснить, когда ее вызывать.

Для начала добавим функцию `log_request` в файл `vsearch4web.py`. Вы можете поместить ее в любую часть файла, но мы вставили ее непосредственно перед функцией `do_search` и ее декоратором `@app.route`. Мы собираемся вызвать эту функцию из функции `do_search`, поэтому разумно разместить ее перед вызывающей функцией.

Нам нужно вызвать `log_request` до завершения функции `do_search`, но после записи результата вызова `search4letters` в `results`. Так выглядит код `do_search` после добавления вызова новой функции.

```
    ...
phrase = request.form['phrase']
letters = request.form['letters']
title = 'Here are your results:'
results = str(search4letters(phrase, letters))
log_request(request, results)
return render_template('results.html',
    ...
```

Краткий обзор

Прежде чем продолжить работу с последней версией `vsearch4web.py`, приведите свой код в точное соответствие с нашим. Ниже показано содержимое файла, в котором выделены последние изменения.

```
from flask import Flask, render_template, request
from vsearch import search4letters

app = Flask(__name__)

def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req, res, file=log)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

if __name__ == '__main__':
    app.run(debug=True)
```

Последние изменения, реализующие журналирование всех веб-запросов. Журнал хранится в файле «vsearch.log».

Возможно, вы заметили, что ни в одной функции в веб-приложении нет комментариев. Позволительное упущение в нашем случае (ведь нам нужно уместить все на нескольких страницах, а это иногда сложно сделать). Обратите внимание: код, доступный для загрузки на веб-сайте книги, содержит подробные комментарии.

Запуск веб-приложения

Запустим эту версию веб-приложения (если необходимо) в командной строке. В *Windows* выполните эту команду:

```
C:\webapps> py -3 vsearch4web.py
```

В *Linux* или *Mac OS X* выполните эту команду:

```
$ python3 vsearch4web.py
```

После запуска приложения попробуем сохранить в журнал данные из HTML-формы.

Пробная поездка

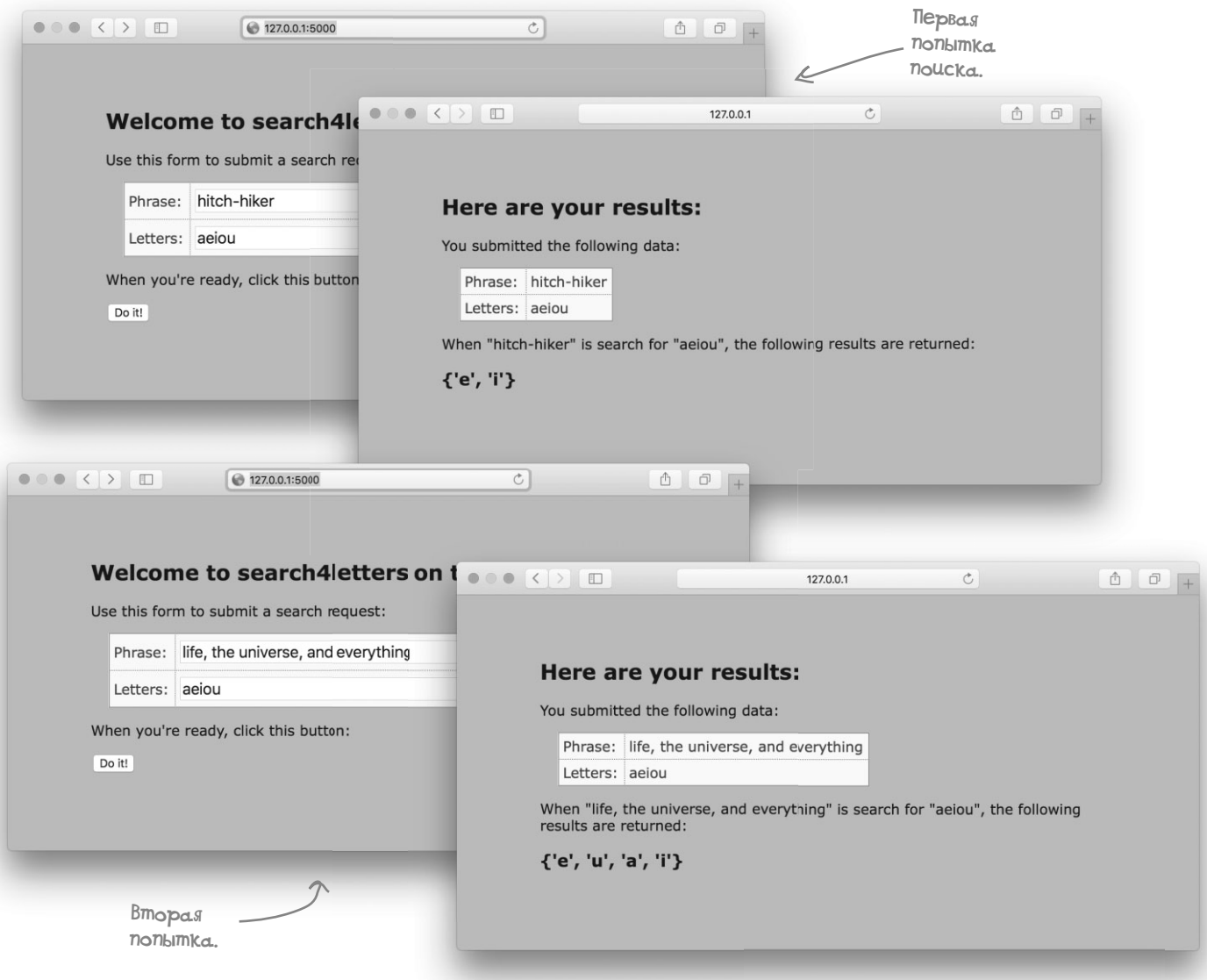
Используя веб-браузер, заполните и отправьте HTML-форму приложения. Если хотите последовать за нами, выполните поиск трижды с использованием следующих значений `phrase` и `letters`:

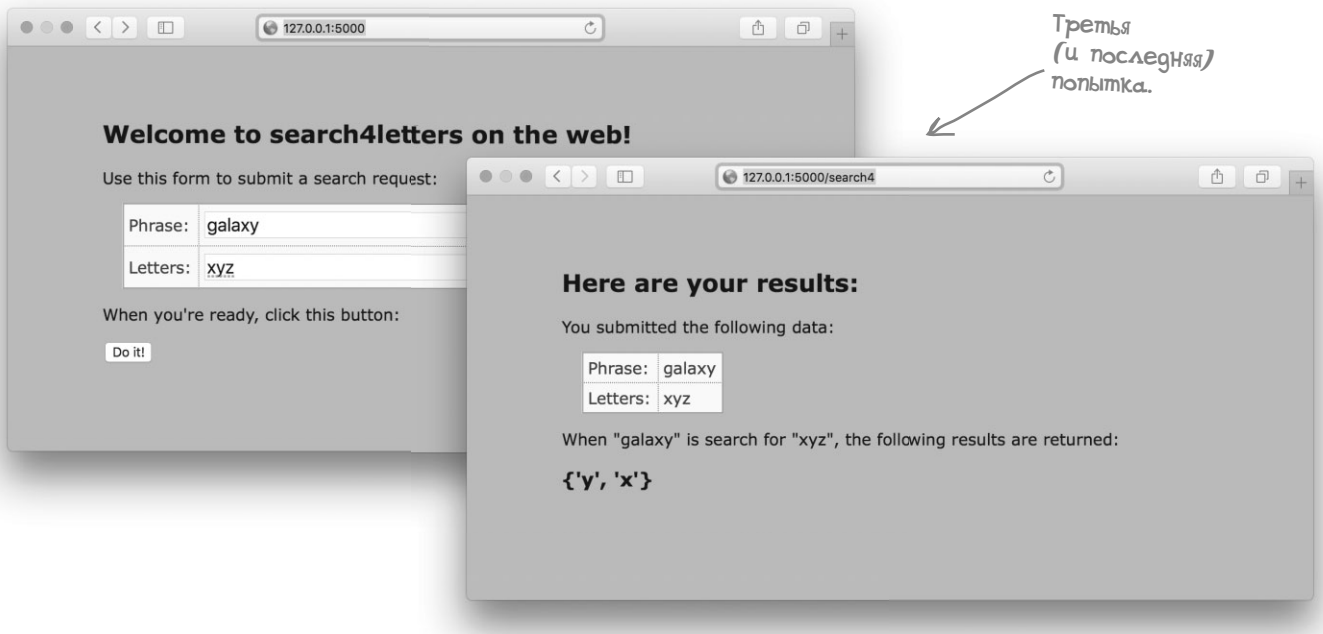
`hitch-hiker` with `aeiou`.

`life, the universe, and everything` with `aeiou`.

`galaxy` with `xyz`.

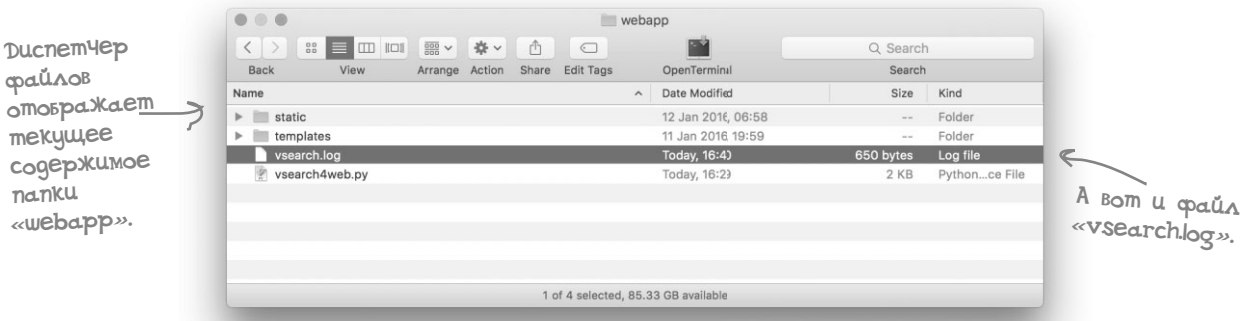
Прежде чем начать, обратите внимание, что файл `vsearch.log` пока отсутствует.





Данные записаны в журнал (в ходе этих попыток)

Каждый раз, когда для отправки данных в веб-приложение используется HTML-форма, функция `log_request` сохраняет детали веб-запроса и результаты поиска в файл журнала. После первой попытки поиска в папке веб-приложения создается файл `vsearch.log`.



Заманчиво воспользоваться текстовым редактором, чтобы заглянуть в файл `vsearch.log`. Но что здесь *интересного*? Мы пишем веб-приложение, поэтому давайте организуем доступ к данным в журнале через само приложение. Тогда нам не придется покидать веб-браузер, чтобы посмотреть данные приложения. Итак, создадим новый URL / `viewlog`, отображающий содержимое журнала.

Просмотр журнала в веб-приложении

Добавим в веб-приложение URL */viewlog*. Получив запрос с адресом URL */viewlog*, оно откроет файл `vsearch.log`, прочитает все данные из него и перешлет их браузеру.

Сейчас вы знаете почти все, что нужно. Для начала добавим новую строку `@app.route` (ближе к концу файла `vsearch4web.py`, непосредственно перед инструкцией, сравнивающей `__name__` и `'__main__'`).

```
@app.route('/viewlog')
```

Мы добавили новый URL.

Определив URL, напишем функцию для его обработки. Назовем нашу новую функцию `view_the_log`. Эта функция не имеет параметров и возвращает строковое значение, включающее все строки из файла `vsearch.log`. Вот так выглядит определение функции.

```
def view_the_log() -> str:
```

Мы объявили новую функцию, которая (согласно аннотации) возвращает строку.

Теперь напишем тело функции. Нам нужно открыть файл *для чтения*. Этот режим функция `open` использует по умолчанию, поэтому достаточно передать ей только имя файла. Для управления контекстом обработки файла используем инструкцию `with`.

```
with open('vsearch.log') as log:
```

Открыть файл журнала для чтения.

В теле инструкции `with` нужно прочитать содержимое файла. Возможно, вы сразу подумали об организации цикла по строкам в файле. Однако интерпретатор предоставляет метод `read`, который возвращает все содержимое файла за один раз. Следующая, единственная строка кода именно это и делает, создавая новый строковый объект `contents`.

```
contents = log.read()
```

Читаем файл целиком и присваиваем его содержимое переменной (с именем «contents»).

Прочитав файл, инструкция `with` завершается (и закрывает файл). Теперь вы готовы передать данные веб-браузеру. Это просто.

```
return contents
```

Возвращаем список строк «contents» вызывающему коду.

Сложив все вместе, вы можете написать код обработки запроса с URL */viewlog*; и вот как это выглядит.

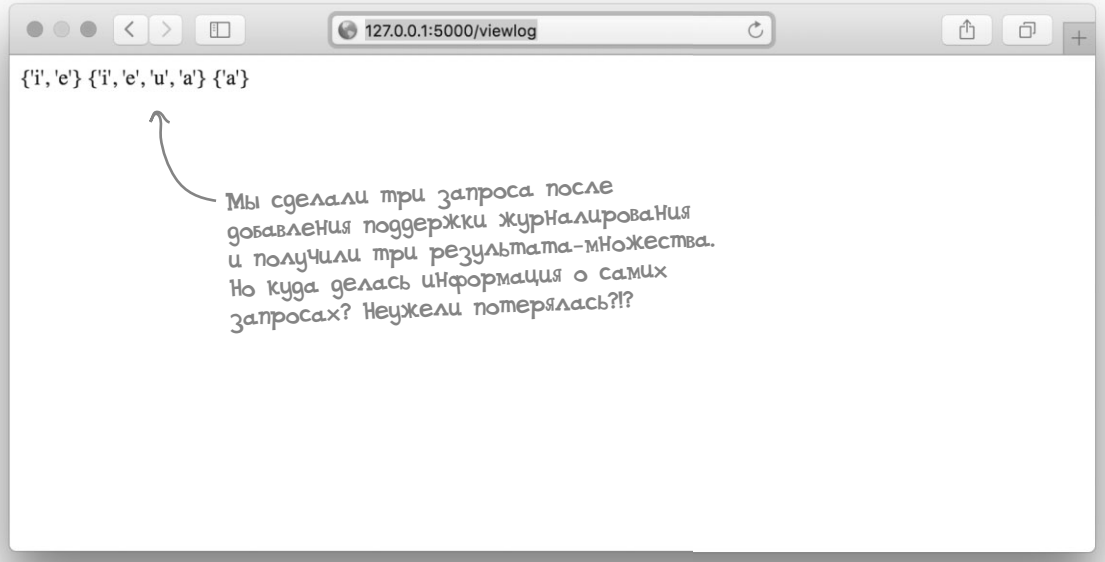
Весь код, необходимый для поддержки URL «/viewlog».

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

Пробная поездка

После добавления нового кода и сохранения веб-приложение должно автоматически перезапуститься. Вы можете сделать несколько запросов, если хотите, но в журнале уже хранится информация о запросах, которые мы сделали несколько страниц назад. Все новые запросы будут добавлены в конец файла. Используем URL */viewlog*, чтобы посмотреть, что сохранилось. Наберите в адресной строке браузера **`http://127.0.0.1:5000/viewlog`**.

Вот что мы увидели в *Safari*, в *Mac OS X* (мы также использовали *Firefox* и *Chrome* и получили такой же результат).



С чего начать, если что-то пошло не так...

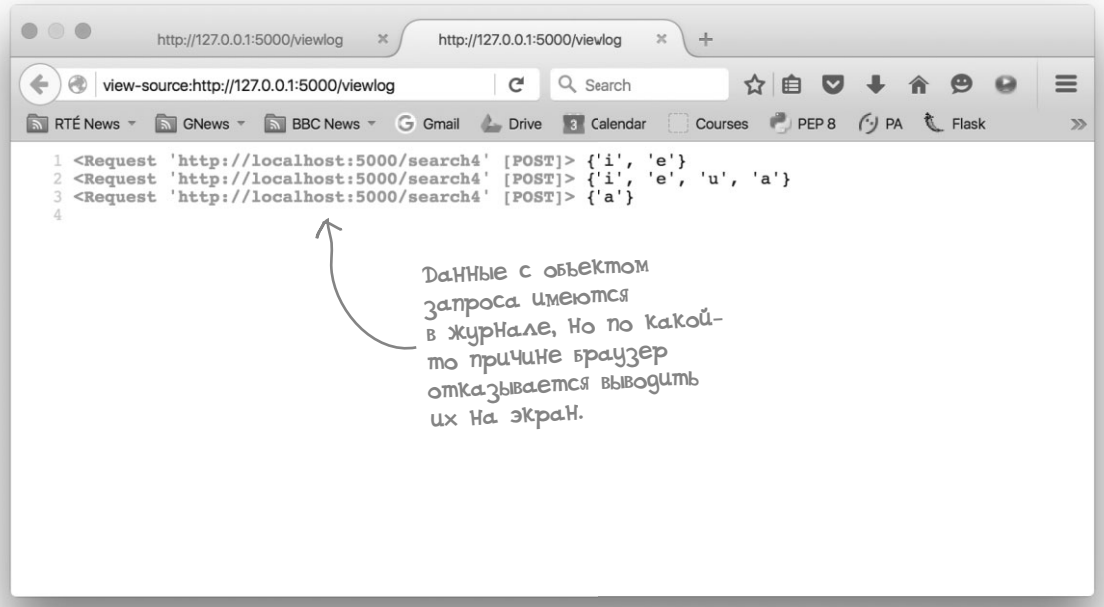
Если вывод приложения не соответствует ожиданиям (как в этом случае), лучше всего начать с проверки данных, присылаемых приложением. Важно отметить, что на экране *отображаются* данные, переданные веб-приложением и интерпретированные браузером. Большинство браузеров позволяют увидеть полученные данные без интерпретации — **исходный код** страницы. Для отладки очень полезно посмотреть на него — это помогает понять, что пошло не так.

Если вы пользуетесь *Firefox* или *Chrome*, щелкните правой кнопкой мыши в окне браузера и выберите пункт **View Page Source** (**Фактический текст в меню**) в контекстном меню, чтобы увидеть исходные данные, полученные от приложения. В *Safari* сначала нужно открыть доступ к инструментам разработчика: в настройках *Safari*, внизу на вкладке **Advanced** (**Дополнения**), установите флажок *Show Develop menu in the menu bar* (*Показывать меню «Разработка» в строке меню*). После этого закройте диалог с настройками, щелкните правой кнопкой мыши в окне браузера и выберите пункт **Show Page Source** (**Фактический текст в меню**) в контекстном меню. Посмотрите на исходный код страницы и сравните его с тем, что получили мы (показан на следующей странице).

Исследуем исходный код страницы

Как вы помните, функция `log_request` сохраняет в журнале две порции данных для каждого запроса: объект запроса и результаты вызова `search4letters`. Но на странице просмотра журнала (с URL */viewlog*) мы видим только результаты. Поможет ли исходный код (то есть данные, полученные от веб-приложения) пролить хоть какой-то свет на произошедшее с объектом запроса?

Вот что мы увидели при просмотре исходного кода страницы в *Firefox*. Тот факт, что каждый объект запроса окрашен в красный цвет, — еще одна подсказка, что с данными что-то не так.



Чтобы понять, почему данные о запросе не выводятся на экран, нужно знать, как браузеры интерпретируют разметку HTML, и тот факт, что *Firefox* выделил данные запроса красным, помогает понять произошедшее. Оказывается, с данными ничего не произошло. Но угловые скобки (< и >) вводят браузер в заблуждение. Когда встречается открывающая угловая скобка, браузер считает, что все содержимое между нею и соответствующей закрывающей угловой скобкой является HTML-тегом. Поскольку `<Request>` не является допустимым HTML-тегом, современные браузеры игнорируют его и не выводят текст между угловыми скобками на экран, что, собственно, и произошло. Загадка исчезновения данных запроса решена. Но мы же должны показать их на странице с URL */viewlog*.

Для этого нужно сообщить браузеру, что он не должен воспринимать угловые скобки, окружающие объект запроса, как HTML-тег, а обрабатывать их как обычный текст. Нам повезло, потому что в фреймворке Flask есть функция, которая нам поможет.

Пришло время экранировать (ваши данные)

Создатели стандарта HTML понимали, что иногда разработчикам веб-страниц может понадобиться отобразить угловые скобки (и другие символы, имеющие специальное значение в HTML). Соответственно они выдвинули идею, которая называется *экранирование* (*escaping*): кодирование специальных символов HTML так, чтобы они могли появляться на странице, но не воспринимались как разметка HTML. Для этого каждому специальному символу были определена своя последовательность. Это очень простая идея: специальный символ, такой как <, кодируется как `<`, а > — как `>`. Если отправить такие последовательности *вместо* исходных данных, браузер сделает то, что нужно: отобразит < и >, а также весь текст между ними.

В фреймворке Flask есть функция `escape` (которая фактически заимствуется из JinJa2). Если передать ей некоторые исходные данные, она переведет их в экранированный HTML-эквивалент. Поэкспериментируем с `escape` в интерактивной оболочке >>> Python, чтобы понять, как она работает.

Для умников

Объект **Markup** — это текст, который можно безопасно использовать в контексте HTML/XML. Markup наследует свойства и методы встроенных в Python строк *Юникода* и может использоваться как строка.

Для начала импортируем функцию `escape` из модуля `flask`, а затем вызовем `escape` со строкой, не содержащей специальных символов.

Импорт функции.

```
>>> from flask import escape
>>> escape('This is a Request')
Markup('This is a Request')
```

Вызов «escape» с нормальной строкой.

Нет изменений.

Функция `escape` возвращает *объект* `Markup`, который действует как строка во всех смыслах. Если передать функции `escape` строку, содержащую какие-либо специальные символы HTML, она заменит их, как показано ниже.

Вызвать «escape» со строкой, содержащей специальные символы.

```
>>> escape('This is a <Request>')
Markup('This is a &lt;Request&gt;')
```

Специальные символы были преобразованы (экранированы).

Как показано в предыдущем примере, вы можете считать объект `Markup` обычной строкой.

Организовав вызов функции `escape` и передачу ей данных из файла журнала, мы решим проблему отображения на экране данных о запросе. Это несложно, потому что файл журнала читается «в один прием» внутри функции `view_the_log` и возвращается как строка.

Сюда сохраняется содержимое файла журнала (как строка).

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

Чтобы решить проблему, мы должны вызвать `escape` с аргументом `contents`.

Просмотр всего журнала в веб-приложении

Нам нужно внести в код очень простое, но крайне результативное изменение. Добавьте `escape` в список импорта из модуля `flask` (в начале программы), а затем вызовите `escape` при возврате строки из метода `join`:

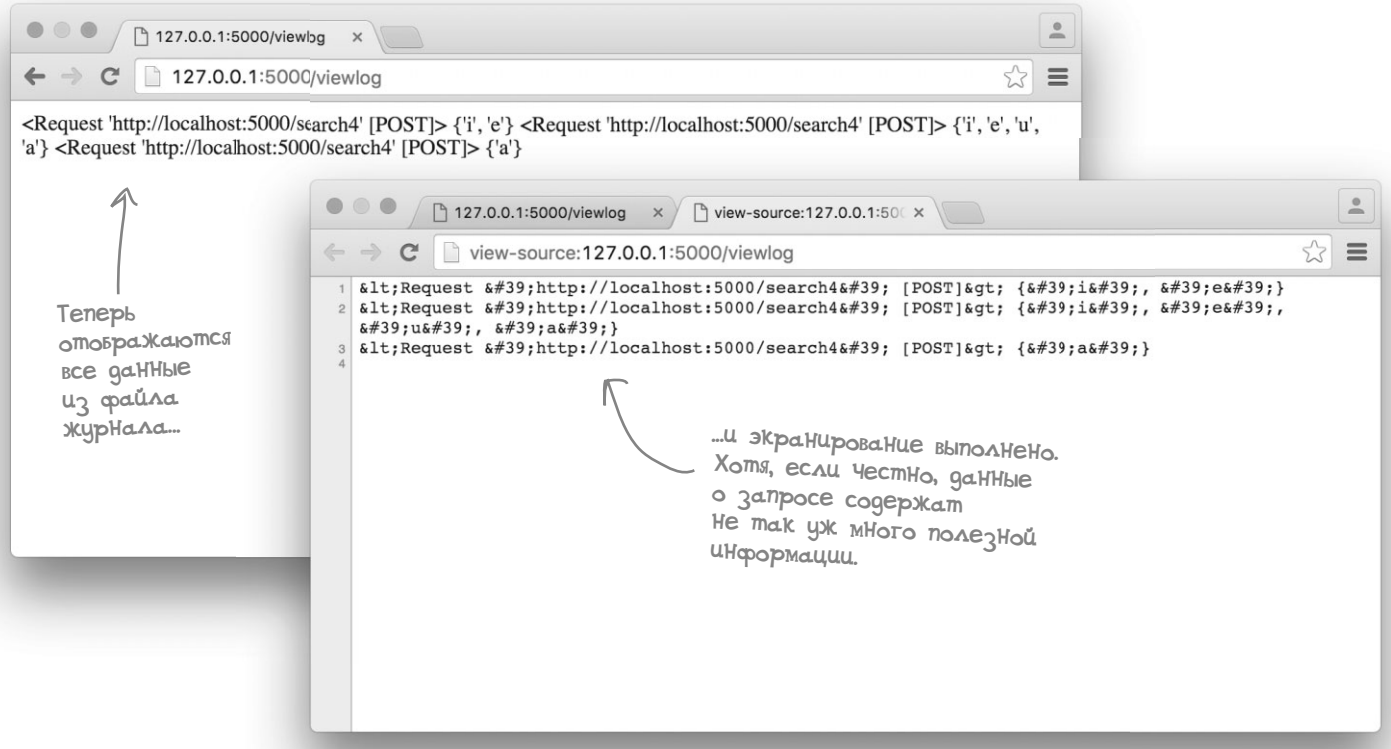
```
from flask import Flask, render_template, request, escape
      ...
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
       contents = log.read()
    return escape(contents)
```

Добавить в список импорта.

Вызвать «escape» и передать ей возвращаемую строку.

Пробная поездка

Измените программу, добавив импорт и вызов `escape`, как показано выше, и сохраните код (чтобы перезапустить веб-приложение). Затем перезагрузите URL */viewlog* в браузере. Теперь на экране должны отображаться все данные из журнала. Загляните в исходный код страницы, чтобы убедиться, что экранирование действительно выполнено. Вот что увидели мы при тестировании версии веб-приложения в *Chrome*.



Узнаем больше об объекте запроса

Данные в файле журнала, относящиеся к запросу, в действительности почти не несут полезной информации. Вот пример текущего содержимого журнала; даже притом что результаты отличаются, сами веб-запросы выглядят *абсолютно* одинаковыми.

Все веб-запросы выглядят одинаково.

```
<Request 'http://localhost:5000/search4' [POST]> {'i', 'e'}
<Request 'http://localhost:5000/search4' [POST]> {'i', 'e', 'u', 'a'}
<Request 'http://localhost:5000/search4' [POST]> {'a'}
```

Результаты отличаются.

Мы сохраняем каждый веб-запрос на уровне объекта, хотя на самом деле нужно было заглянуть *внутрь* запроса и сохранить данные, которые он содержит. Чтобы узнать, что содержит тот или иной объект, нужно передать его встроенной функции `dir`, которая вернет список методов и атрибутов этого объекта.

Внесем небольшое изменение в функцию `log_request`, чтобы сохранить в журнале результат вызова `dir` для каждого объекта запроса. Это очень небольшое изменение... вместо самого объекта `req` передадим функции `print` в первом аргументе строковую версию результата вызова `dir(req)`. Ниже приведена новая версия функции `log_request` с выделенным изменением.

```
def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(str(dir(req)), res, file=log)
```

Вызов «dir» с аргументом «req» вернет список, который мы превращаем в строку, передав его в функцию «str». Полученная строка затем сохраняется в файл журнала вместе со значением «res».

Упражнение

Попробуем запустить обновленный код и посмотрим, как изменится результат. Для этого выполните следующие шаги.

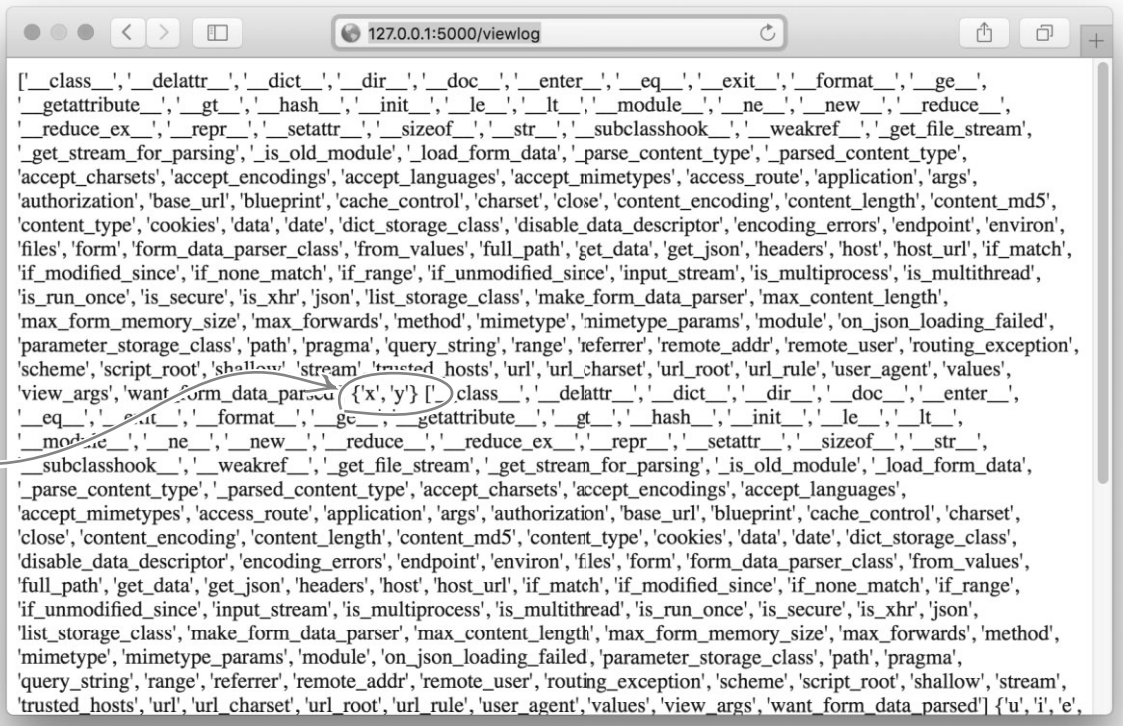
1. Измените свою функцию `log_request`, чтобы она соответствовала нашей функции.
2. Сохраните `vsearch4log.py`, чтобы веб-приложение перезапустилось.
3. Найдите и удалите файл `vsearch.log`.
4. С помощью браузера выполните поиск несколько раз.
5. Посмотрите новый файл журнала с помощью URL */viewlog*.

Внимательно посмотрите, что появилось в браузере. Как нам это поможет?

Пробная поездка

Вот что мы видим после выполнения пяти пунктов с предыдущей страницы. Мы используем браузер *Safari* (хотя то же самое покажет любой другой браузер).

Похоже на беспорядочную мешанину. Но посмотрите внимательнее — вот результат одной из попыток поиска, которые мы выполнили.



И что со всем этим делать?

Приложив некоторые усилия, можно выделить результаты работы запроса. Остальное — это результат вызова `dir` с объектом запроса. Как видите, каждый запрос имеет много методов и атрибутов (даже без учета *внутренних* переменных и методов). Не нужно сохранять в журнал *все* эти атрибуты.

Мы исследовали атрибуты и решили сохранять три из них, которые посчитали важными для журналирования:

`req.form` — данные из HTML-формы веб-приложения.

`req.remote_addr` — IP-адрес веб-браузера, приславшего форму.

`req.user_agent` — строка, идентифицирующая браузер пользователя.

Отредактируем функцию `log_request` так, чтобы она сохраняла эти три параметра и результаты вызова `search4letters`.

Журналирование отдельных атрибутов веб-запроса

Нам нужно записать в журнал четыре блока данных — данные из формы, IP-адрес клиента, идентификационную строку браузера и результаты вызова `search4letters`. В результате первой попытки изменить `log_request` может получиться примерно такой код, сохраняющий каждый блок данных отдельным вызовом `print`.

```
def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, file=log)
        print(req.remote_addr, file=log)
        print(req.user_agent, file=log)
        print(res, file=log)
```

Сохраним каждый блок данных отдельным вызовом «print».

Код работает, но есть проблема. Каждый вызов `print` по умолчанию добавляет символ переноса строки, то есть для каждого запроса в файл журнала записываются **четыре** отдельные строки. Вот как выглядят данные, записанные этим кодом.

Данные, введенные в HTML-форму, тоже появляются на отдельной строке. Кстати, «ImmutableMultiDict» — это особая версия словаря в фреймворке Flask (и работает как словарь).

Отдельная строка для IP-адреса.

```
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'hitch-hiker')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'i', 'e'}
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'life, the universe, and everything')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'a', 'e', 'i', 'u'}
ImmutableMultiDict([('letters', 'xyz'), ('phrase', 'galaxy')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'x', 'y'}
```

Идентификационная информация о браузере тоже отдельной строкой.

Ясно видимые результаты вызова «search4letters» (и снова отдельная строка).

Эта стратегия не имеет фатальных недостатков (сохраненные данные легко читаются). Но представьте, что нужно читать эти данные внутри программы. Для каждого веб-запроса придется прочитать из файла журнала **четыре** строки — по одной на каждый элемент сохраненных данных, хотя все **четыре** строки относятся к *одному* запросу. Такой подход кажется непрактичным и избыточным. Гораздо удобнее читать всего **одну** строку для каждого веб-запроса.

Журналирование данных в одну строку с разделителями

Лучшая стратегия журналирования — записывать все элементы данных в одну строку и отделять один элемент от другого с помощью подходящего разделителя.

Выбрать разделитель непросто, потому что этот символ не должен встречаться в сохраняемых данных. Использовать пробел в качестве разделителя бесполезно (в сохраняемых данных содержится много пробелов), и даже двоеточие (`:`), запятая (`,`) и точка с запятой (`;`) в качестве разделителей не подходят. Мы посоветовались с программистами из лаборатории *Head First* и решили использовать в качестве разделителя символ вертикальной черты (`|`): его легко заметить, и он редко бывает частью данных. Воспользуемся этим советом и посмотрим, что получится.

Как мы видели выше, поведение функции `print` по умолчанию можно изменить, передав ей дополнительные аргументы. Кроме аргумента `file`, есть аргумент `end`, в котором можно передать альтернативный символ *конца строки*, отличный от символа переноса.

Изменим `log_request` таким образом, чтобы вместо символа переноса строки по умолчанию в конце строки использовался символ вертикальной черты.

Для умников

Рассматривайте **разделитель** как последовательность из одного или нескольких символов, выполняющую функцию границ в строке текста. Классический пример — символ запятой (`,`), который используется в CSV-файлах. (Аббревиатура CSV расшифровывается как «Comma Separated Value» — значения, разделенные запятыми. — *Прим. науч. ред.*)

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, file=log, end='|')
        print(req.remote_addr, file=log, end='|')
        print(req.user_agent, file=log, end='|')
        print(res, file=log)
```

Каждая из этих инструкций «print» вместо символа переноса строки, который используется по умолчанию, выведет вертикальную черту.

Все получилось, как мы хотели: каждый веб-запрос теперь сохраняется в единственной строке и отдельные блоки данных разделены символом вертикальной черты. Вот как выглядят данные в файле журнала после изменения `log_request`:

Каждый веб-запрос записан в одну строку (мы разнесли его по нескольким строкам, чтобы уместить по ширине страницы).

```
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'hitch-hiker')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2
Safari/601.3.9|{'e', 'i'}
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'life, the universe, and everything')])|12
7.0.0.1|Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko)
Version/9.0.2 Safari/601.3.9|{'e', 'u', 'a', 'i'}
ImmutableMultiDict([('letters', 'xyz'), ('phrase', 'galaxy')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2
Safari/601.3.9|{'y', 'x'}
```

Заметили? Символы вертикальной черты используются как разделители. Три вертикальные черты означают, что в одной строке сохранены четыре элемента данных.

Выполнены три веб-запроса, поэтому в файле журнала три строки с данными.

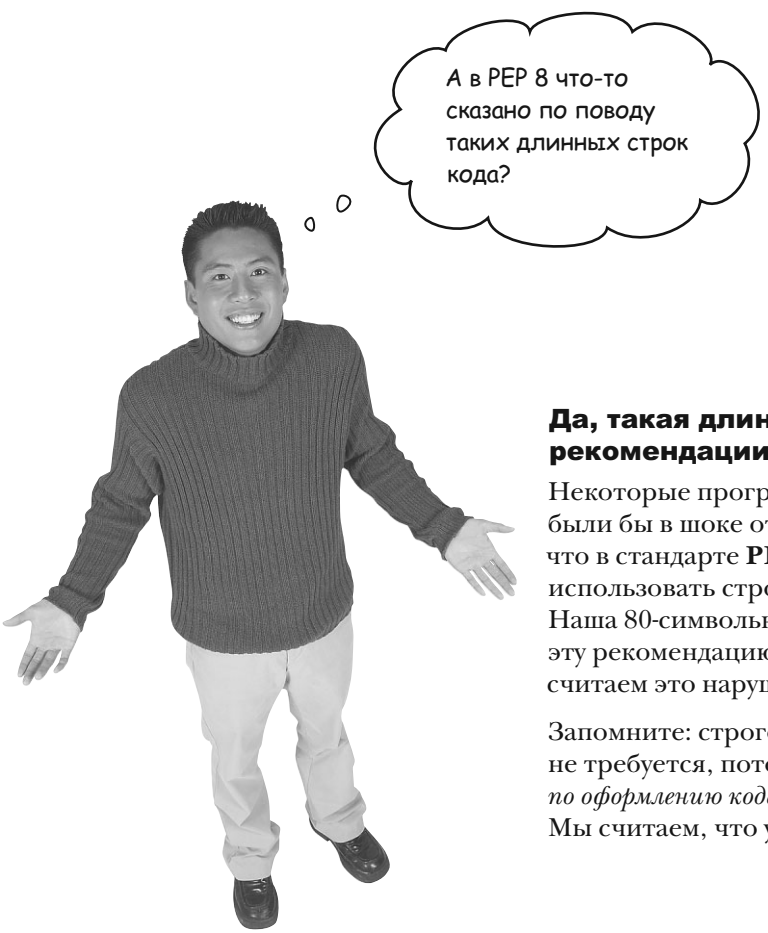
Последнее изменение в коде, выполняющем журналирование

Излишне подробный код — больное место многих программистов на Python. Последняя версия `log_request` работает хорошо, но в ней больше кода, чем требуется. Например, для записи каждого элемента данных используется отдельный вызов `print`.

Функция `print` имеет еще один необязательный аргумент, `sep`, позволяющий указать разделитель для вывода нескольких значений в одном вызове `print`. По умолчанию `sep` соответствует одному пробельному символу, но можно использовать любое значение. В следующем коде мы заменили четыре вызова `print` (с предыдущей страницы) одним, в котором передали в аргументе `sep` символ вертикальной черты. Так мы избавились от необходимости указывать значение для параметра `end`, поэтому все упоминания `end` можно удалить из кода.

Единственный вызов «print» вместо четырех.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```



Да, такая длинная строка кода нарушает рекомендации PEP 8.

Некоторые программисты на Python были бы в шоке от последней строки, потому что в стандарте **PEP 8** не рекомендуется использовать строки, длиннее 79 символов. Наша 80-символьная строка *немного* нарушает эту рекомендацию, но, учитывая нашу цель, мы считаем это нарушение допустимым.

Запомните: строгое следование PEP 8 не требуется, потому что PEP 8 — *руководство по оформлению кода*, а не свод нерушимых правил. Мы считаем, что у нас все в порядке.

Упражнение

Посмотрим, что изменится после запуска нового кода. Исправьте свою функцию `log_request`, как показано ниже.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

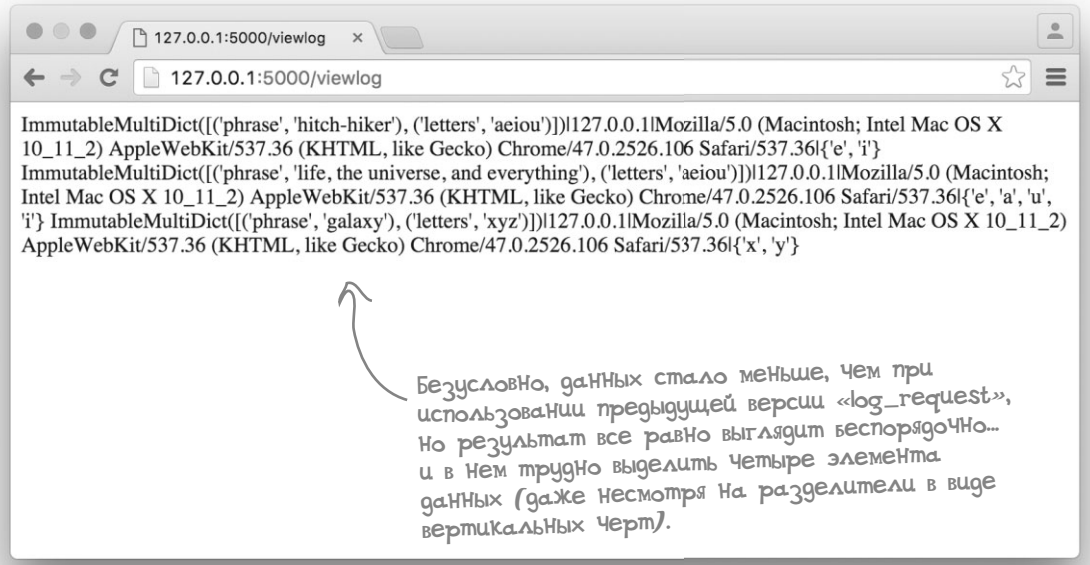
Затем выполните эти четыре шага.

1. Сохраните `vsearch4log.py` (чтобы перезапустить веб-приложение).
2. Найдите и удалите текущую версию файла `vsearch.log`.
3. Выполните несколько попыток поиска в браузере.
4. Запросите новый файл журнала, обратившись по адресу URL */viewlog*.

Еще раз внимательно посмотрите на содержимое журнала в браузере. Стал ли он лучше, чем был?

Пробная поездка

Выполнив четыре шага, перечисленные в упражнении выше, в *Chrome*, мы увидели на экране:



Вывод данных в читаемом формате

Браузер отобразил данные в исходном, *неформатированном виде*. Как вы наверняка помните, мы экранировали специальные символы HTML после чтения данных из журнала, и это все, что мы сделали перед передачей строки веб-браузеру. Современные веб-браузеры получают строку, удаляют нежелательные пробельные символы (например, лишние пробелы, переносы строк и т. д.), а затем выводят результат на экран. Именно это и произошло в нашей «Пробной поездке». Видны все данные из журнала, но их сложно читать. Нужно дополнительно обработать исходные данные (чтобы они легко читались), а лучший способ сделать вывод более читаемым — вывести данные в виде таблицы.

```
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106
Safari/537.36|{'e', 'i'} ImmutableMultiDict([('phrase', 'life, the universe, and
everything'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'e', 'a', 'u',
'i'} ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106
Safari/537.36|{'x', 'y'}
```

Можно ли взять эти (нечитаемые) исходные данные...

...и превратить их в таблицу, которая выглядит так?

Form Data	Remote_addr	User_agent	Results
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36	{'e', 'i'}
ImmutableMultiDict([('phrase', 'life, the universe, and everything'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36	{'e', 'a', 'u', 'i'}
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36	{'x', 'y'}

Если бы наше веб-приложение могло выполнить такое превращение, тогда *любой* мог бы посмотреть содержимое журнала в браузере и, возможно, эта информация имела бы для него смысл.

Вам это ничего не напоминает?

Еще раз взгляните на то, что нам нужно получить. Чтобы не занимать много места, мы показали только верхушку таблицы. То, что мы сейчас пытаемся получить, не напоминает вам ничего из того, что мы изучали ранее?

Form Data	Remote_addr	User_agent	Results
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36	{'e', 'i'}

Поправьте меня, если я ошибаюсь, но разве это не та же структура данных, которую мы видели в конце главы 3?

Да. Это похоже на то, что мы видели раньше.

В конце главы 3 мы превратили таблицу, показанную ниже, в сложную структуру данных — словарь словарей.

Имя	Пол	Род занятий	Планета
Форд Префект	Мужской	Исследователь	Бетельгейзе 7
Артур Дент	Мужской	Приготовление бутербродов	Земля
Триша МакМиллан	Женский	Математик	Земля
Марвин	Неизвестен	Робот-параноик	Неизвестна

Своей формой эта таблица очень похожа на то, что мы собираемся получить сейчас, но нужно ли использовать здесь словарь словарей?

Использовать словарь словарей... или еще что-то?

Таблица с данными из главы 3 хорошо укладывается в модель словаря словарей, потому что позволяет легко углубиться в структуру данных и выбрать требуемое поле. Например, чтобы узнать родную планету Форда Префекта, достаточно использовать конструкцию:

```
people['Ford']['Home Planet']
```

Получить доступ к данным Форда...

...затем выбрать значение, связанное с ключом «Home Planet».

Когда нужен произвольный доступ к структуре данных, нет ничего лучше, чем словарь словарей. Но подходит ли он для представления данных из журнала?

Давайте подумаем, что у нас есть.

Внимательно посмотрим на данные в журнале

Как вы помните, что каждая строка в журнале состоит из четырех элементов данных, разделенных вертикальной чертой: данные из HTML-формы, IP-адрес клиента, строка идентификации веб-браузера, а также результаты вызова `search4letters`.

Вот пример строки данных из файла `vsearch.log`, где выделены все символы вертикальной черты.

Данные из формы.

IP-адрес удаленного компьютера.

```
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'x', 'y'}
```

Строка идентификации браузера.

Результаты вызова «search4letters».

Данные, прочитанные из файла журнала `vsearch.log`, попадают в код в виде *списка строк*, благодаря использованию метода `readlines`. Вам едва ли потребуется произвольный доступ к отдельным элементам данных из журнала, поэтому их преобразование в словарь словарей — не самая удачная идея. Однако вам потребуется обрабатывать строки *по порядку* и так же *по порядку* обрабатывать в них элементы данных. У вас есть список строк, а значит, вы практически готовы начать обработку данных в цикле `for`. Однако сейчас строка данных — единое целое, и это проблема. Обрабатывать строку было бы проще, если бы она была списком элементов, а не одной длинной строкой. Возникает вопрос: *можно ли преобразовать строку в список*?

То, что объединено, может быть разбито

Мы знаем, что список строк можно превратить в одну строку, добавив разделители. Давайте еще раз посмотрим, как это делается в интерактивной оболочке >>>.

Список отдельных строк.

```
>>> names = ['Terry', 'John', 'Michael', 'Graham', 'Eric']
>>> pythons = '|'.join(names)
>>> pythons
'Terry|John|Michael|Graham|Eric'
```

Операция объединения.

Единая строка, в которой каждая подстрока из списка «names» отделена от следующей вертикальной чертой.

Благодаря операции объединения список строк превратился в одну строку, в которой каждый элемент отделен от другого вертикальной чертой (в нашем случае). Эту строку можно снова превратить в список с помощью метода `split`, доступного в каждой строке Python.

Превратить строку в список, разбив ее по заданному разделителю.

```
>>> individuals = pythons.split('|')
>>> individuals
['Terry', 'John', 'Michael', 'Graham', 'Eric']
```

Мы вернули список строк.

Получение списка списков из списка строк

Теперь, когда в нашем арсенале появился метод `split`, вернемся к данным, хранящимся в файле журнала, и подумаем, что нужно с ними сделать. В настоящий момент каждая строка в файле `vsearch.log` имеет следующий вид.

Исходные данные.

```
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'x', 'y'}
```

Сейчас веб-приложение читает все строки из файла `vsearch.log` в список с именем `contents`. Ниже показаны последние три строки кода в функции `view_the_log`, которые читают данные из файла и преобразуют их в очень длинную строку.

```
    ...
with open('vsearch.log') as log:
    contents = log.readlines()
return escape(''.join(contents))
```

Открыть файл журнала...

...и прочитать все строки данных в список с именем «contents».

Последняя строка в функции `view_the_log` принимает список строк `contents` и объединяет их в одну большую строку (вызовом `join`). Затем полученная единая строка возвращается веб-браузеру.

Если бы переменная `contents` была списком *списков*, а не списком *строк*, появилась бы возможность обрабатывать `contents` *по порядку* в цикле `for`. Тогда можно было бы создать более читаемый вывод, чем тот, что мы сейчас видим на экране.

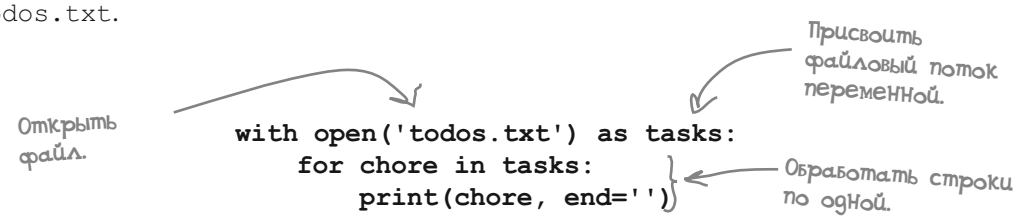
Когда должно происходить преобразование?

На данный момент функция `view_the_log` читает все данные из файла журнала в список строк (`contents`). Но нам надо получить данные в виде списка списков. Вопрос: когда лучше всего выполнять преобразование? Должны ли мы прочитать все данные в список строк, а затем преобразовать его в список списков или лучше строить список списков непосредственно при чтении каждой строки данных?

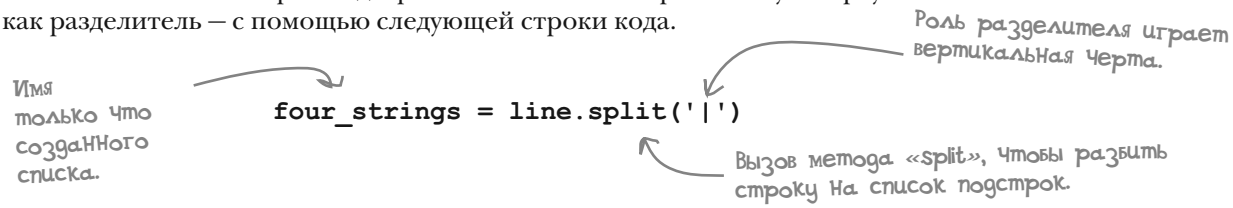
Тот факт, что данные уже находятся в `contents` (благодаря использованию метода `readlines`), означает, что мы уже *один раз* обработали их. Для нас вызов `readlines` — это всего один вызов, а вот интерпретатор (пока выполняет `readlines`) выполняет цикл по всем данным в файле. Если мы еще раз пройдем по данным (чтобы преобразовать строки в списки), мы **удвоим** количество итераций. Это не так важно, если в журнале немного строк... но превращается в большую проблему, когда журнал вырастет до значительных размеров. Вывод: *если можно сделать всю обработку за один проход – давайте сделаем это!*

Обработка данных: что мы уже знаем

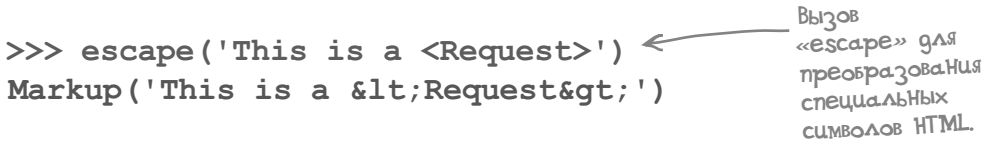
Мы уже видели в этой главе три строки кода, которые обрабатывали файл `todos.txt`.



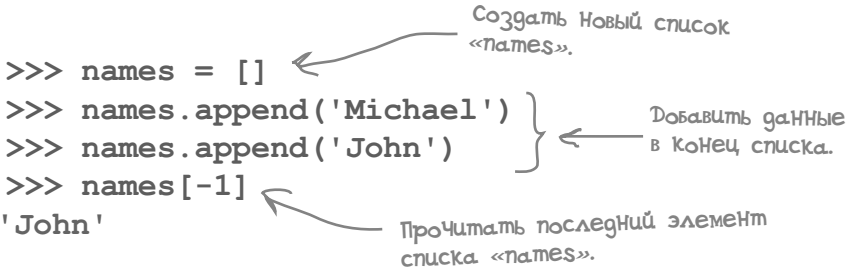
Мы также видели метод `split`, который принимает строку и превращает ее в список строк, разбивая по определенному разделителю (по умолчанию это пробел, если не указано другое). В наших данных разделитель — вертикальная черта. Представьте, что строка данных из журнала хранится в переменной `line`. Мы можем превратить строку `line` в список из четырех подстрок — использовав вертикальную черту как разделитель — с помощью следующей строки кода.



Поскольку при чтении данных из файла мы не можем быть уверены, что там не попадутся символы, имеющие специальное значение в HTML, мы использовали функцию `escape`. Она входит в состав фреймворка Flask и преобразует все специальные символы HTML в строке в экранированные последовательности.



В главе 2 вы узнали, что новый список можно создать, присвоив переменной пустой список (`[]`). Вы также знаете, что с помощью метода `append` можно дописывать новые значения в конец существующего списка, а последний элемент любого списка доступен по индексу `[-1]`.



Посмотрим, помогут ли эти знания завершить упражнение на следующей странице.

Заточите карандаш

Вот как выглядит функция `view_the_log` сейчас.

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

Она читает данные из файла в список строк. Ваша задача — изменить функцию так, чтобы она читала данные в список списков.

При этом данные в списке списков должны быть экранированы, потому что мы не хотим, чтобы там появились специальные символы HTML.

Кроме того, новый код все так же должен возвращать браузеру данные в виде одной строки.

Мы начали, а вам нужно добавить пропущенный код.

Первые две строки остаются без изменений.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    return str(contents)
```

Добавьте сюда новый код.

Функция должна возвращать строку.

Потратьте время. Поэкспериментируйте сначала в интерактивной оболочке >>>. Если на чем-то застрянете, не переживайте – переверните страницу и посмотрите решение.

Заточите карандаш
Решение

Вот как выглядит функция `view_the_log` сейчас.

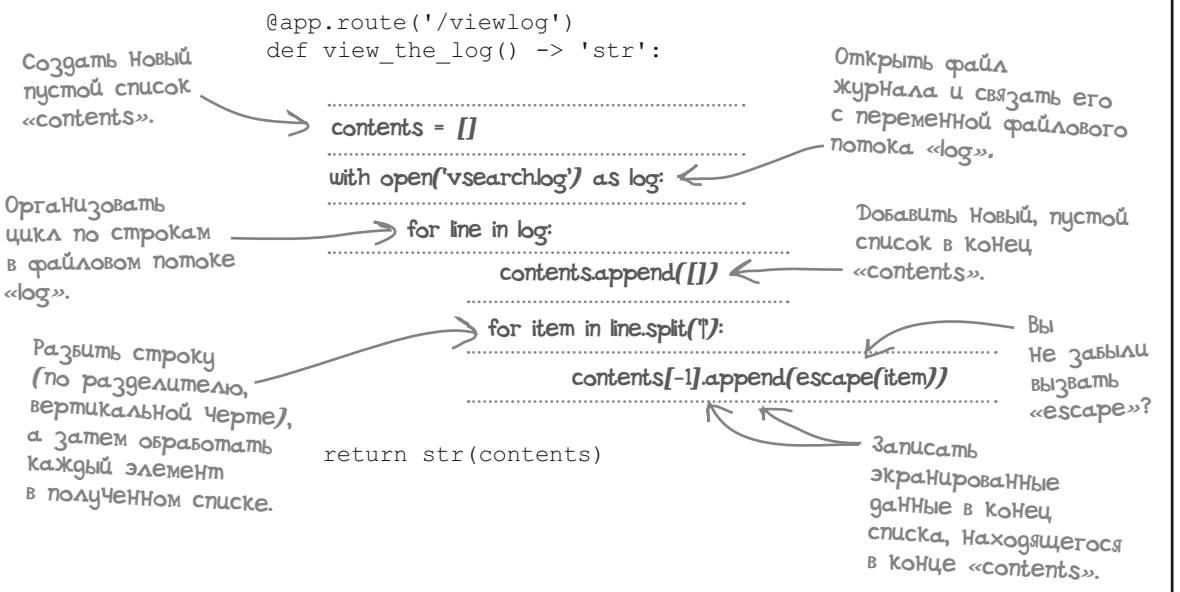
```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

Ваша задача — изменить функцию так, чтобы она читала данные в список списков.

Вы должны были экранировать специальные символы HTML в списке списков, потому что мы не хотим, чтобы они там появлялись.

Кроме того, новый код все так же должен возвращать браузеру данные в виде одной строки.

Мы начали, а вам нужно было только добавить пропущенный код:



Не беспокойтесь, если эта строка кода из функции `view_the_log` вызовет у вас головокружение.

```
contents[-1].append(escape(item))
```

Читайте код изнутри наружу и справа налево.

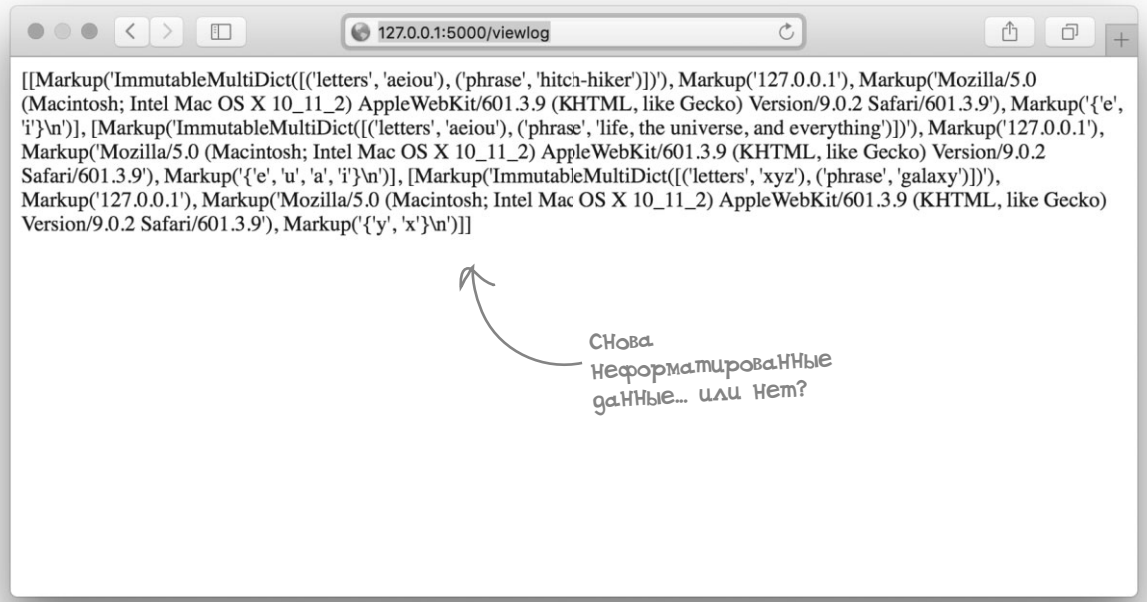
Чтобы понять эту (кажущуюся страшной) строку, попробуйте прочитать ее изнутри наружу и справа налево. Сначала элемент `item` — переменная цикла `for` — передается в функцию `escape`. Полученная в результате строка добавляется в конец списка (`[-1]`) `contents`. Помните: `contents` сам по себе является *списком списков*.

Пробная поездка

Измените функцию `view_the_log` у себя, чтобы она выглядела так:

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Сохраните код (что приведет к перезапуску веб-приложения), а затем перезагрузите URL */viewlog* в браузере. Вот что мы увидели.



Еще раз взглянем на вывод программы

Кажется, вывод, произведенный новой версией `view_the_log`, выглядит так же, как в прошлый раз. Но это не так: сейчас на экран выводится список списков, а не список строк. Это существенное изменение. Если организовать обработку `contents` с помощью специального шаблона Jinjia2, то можно получить более читаемый вывод.

Генерируем читаемый вывод с помощью HTML

Как вы помните, наша цель — получить читаемый вывод, который на экране будет выглядеть лучше, чем данные, полученные в конце предыдущей страницы. В HTML есть набор тегов, определяющих содержимое таблицы, включая `<table>`, `<th>`, `<tr>` и `<td>`. Учитывая это, взглянем еще раз на таблицу, которую мы хотим получить. Для каждой записи из журнала в ней отводится одна строка, организованная в четыре колонки (каждая со своим заголовком).

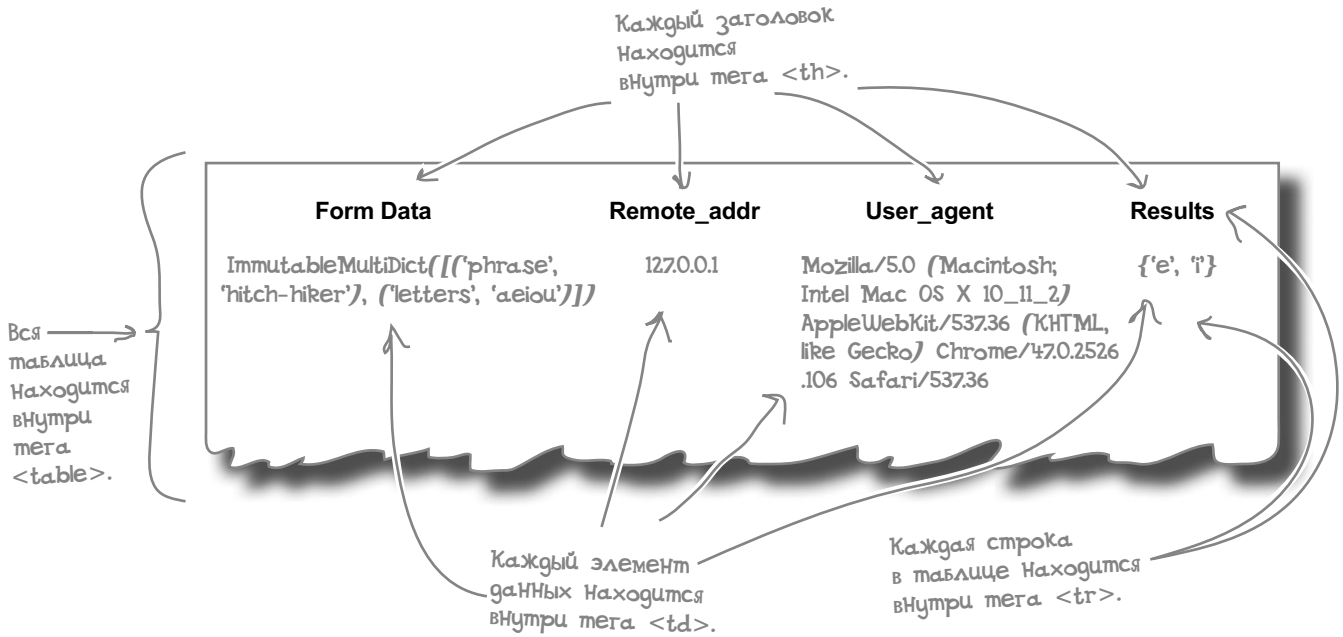
Нам нужно вложить всю таблицу в HTML-тег `<table>`, а каждую строку данных — в отдельный тег `<tr>`. Заголовки описываются с помощью тегов `<th>`, а каждый элемент данных — с помощью тега `<td>`.

Для умников

Краткий обзор табличных тегов HTML:

`<table>` — таблица;
`<th>` — строка данных в таблице;
`<tr>` — заголовок колонки в таблице;
`<td>` — ячейка таблицы.

Для каждого тега имеется соответствующий закрывающий тег: `</table>`, `</th>`, `</tr>` и `</td>`.



Всякий раз, когда возникает необходимость сгенерировать разметку HTML (особенно `<table>`), вспоминайте о Jinjia2. Механизм шаблонов Jinjia2 разработан специально для создания HTML, и в нем есть несколько простых программных конструкций (основанных на синтаксисе Python), с помощью которых можно автоматизировать любую необходимую логику.

В последней главе вы увидите, как теги `{{и}}`, а также `{% block%}` в Jinjia2 позволяют использовать переменные и блоки HTML в виде аргументов шаблонов. Оказывается, теги `{%` и `%}` более универсальные и могут содержать любые *инструкции* Jinjia2, к которым относится и цикл `for`. На следующей странице вы увидите шаблон, использующий цикл `for` для преобразования списка списков, содержащегося в `contents`, в читаемый формат.

Встраиваем логику отображения в шаблон

Внизу показан новый шаблон `viewlog.html`, который можно использовать для преобразования исходных данных из файла журнала в HTML-таблицу. На вход шаблона, в одном из аргументов, подается список списков `contents`. Мы выделили наиболее важные части шаблона. Обратите внимание, что цикл `for` в Jinjia2 очень похож на аналогичную конструкцию в Python. Но есть две большие разницы.

- Символ двоеточия `(:)` в конце строки с `for` не нужен (потому что тег `%}` действует как разделитель).
- Тело цикла завершается конструкцией `{% endfor %}`, потому что Jinjia2 не поддерживает отступы (поэтому нужен какой-то другой механизм).

Необязательно создавать шаблон вручную. Загрузите его с *http://python.itcarlow.ie/ed2/*.

Как видите, первый цикл `for` предполагает, что исходные данные для него находятся в переменной `the_row_titles`, а второй цикл `for` использует переменную `the_data`. Третий цикл `for` (вложенный во второй) в качестве источника данных использует список элементов.

Готовый код

Чтобы добиться единства представления данных, этот шаблон наследует тот же базовый шаблон, который используется во всех частях нашего приложения.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<table>
    <tr>
        {% for row_title in the_row_titles %}
            <th>{{row_title}}</th>
        {% endfor %}
    </tr>
    {% for log_row in the_data %}
        <tr>
            {% for item in log_row %}
                <td>{{item}}</td>
            {% endfor %}
        </tr>
    {% endfor %}
</table>

{% endblock %}
```

Вся таблица помещена в тег `<table>`.

Заголовки (каждый внутри тега `<th>`) занимают отдельную строку (тег `<tr>`).

Каждый элемент данных заключен в тег `<td>`, каждая строка из файла журнала имеет свой тег `<tr>`.

Сохраните новый шаблон в папку `templates`, которую мы использовали раньше.

Создание читаемого вывода с помощью Jinjia2

Поскольку шаблон `viewlog.html` наследует базовый шаблон `base.html`, нужно не забыть передать аргумент `the_title` (список заголовков колонок) в `the_row_titles` и присвоить `contents` аргументу `the_data`.

Сейчас функция `view_the_log` выглядит так.

Сейчас мы возвращаем браузеру строку.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Нужно вызвать функцию `render_template`, чтобы отобразить шаблон `viewlog.html` и передать ей значения трех ожидаемых аргументов: кортеж заголовков в аргументе `the_row_titles`, список `contents` в аргументе `the_data` и подходящее значение в аргументе `the_title`.

Помня об этом, изменим `view_the_log` (мы выделили изменения).

Запомните: кортеж — это список, доступный только для чтения.

Измените аннотацию, чтобы подчеркнуть, что возвращается разметка HTML (а не строка).

Создать кортеж заголовков.

Вызвать «render_template», передав значения для всех аргументов шаблона.

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Выполните все изменения в своей функции `view_the_log` и сохраните их, чтобы Flask перезапустил веб-приложение. Затем перезагрузите в браузере страницу с URL *http://127.0.0.1:5000/viewlog*.

Пробная поездка

Вот что мы увидели после запуска обновленного приложения. Страница выглядит так же, как и все остальные страницы, поэтому мы уверены, что приложение использует правильный шаблон.

Мы довольны результатом (надеемся, вы тоже), потому что он похож на то, что мы хотели получить: читаемый вывод.

127.0.0.1:5000/viewlog

View Log

Form Data	Remcte_addr	User_agent	Results
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'hitch-hiker')])	127.0 0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2 Safari/601.3.9	{'e', 'i'}
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'life, the universe, and everything')])	127.0 0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2 Safari/601.3.9	{'e', 'u', 'a', 'i'}
ImmutableMultiDict([('letters', 'xyz'), ('phrase', 'galaxy')])	127.0.0.1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2 Safari/601.3.9	{'y', 'x'}

Вывод не только читаемый, он еще и красивый. ☺

Если заглянуть в исходный код этой страницы (щелкнув правой кнопкой мыши и выбрав соответствующий пункт в контекстном меню), то можно увидеть, что каждый элемент данных из журнала находится внутри своего тега `<td>`, каждая строка данных — в своем теге `<tr>`, а таблица целиком заключена в тег `<table>`.

Текущее состояние кода нашего веб-приложения

Теперь прервемся на минутку и посмотрим на код нашего веб-приложения. Хотя в приложение добавлен код для ведения журнала (функции `log_request` и `view_the_log`), оно целиком умещается на одной странице. Вот код `vsearch4web.py`, полностью поместившийся окне редактора IDLE (скриншот позволяет видеть код с подсветкой синтаксиса).

vsearch4web.py - /Users/paul/Desktop/_NewBook/ch06/webapp/vsearch4web.py (3.5.1)

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

app = Flask(__name__)

def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)

if __name__ == '__main__':
    app.run(debug=True)
```

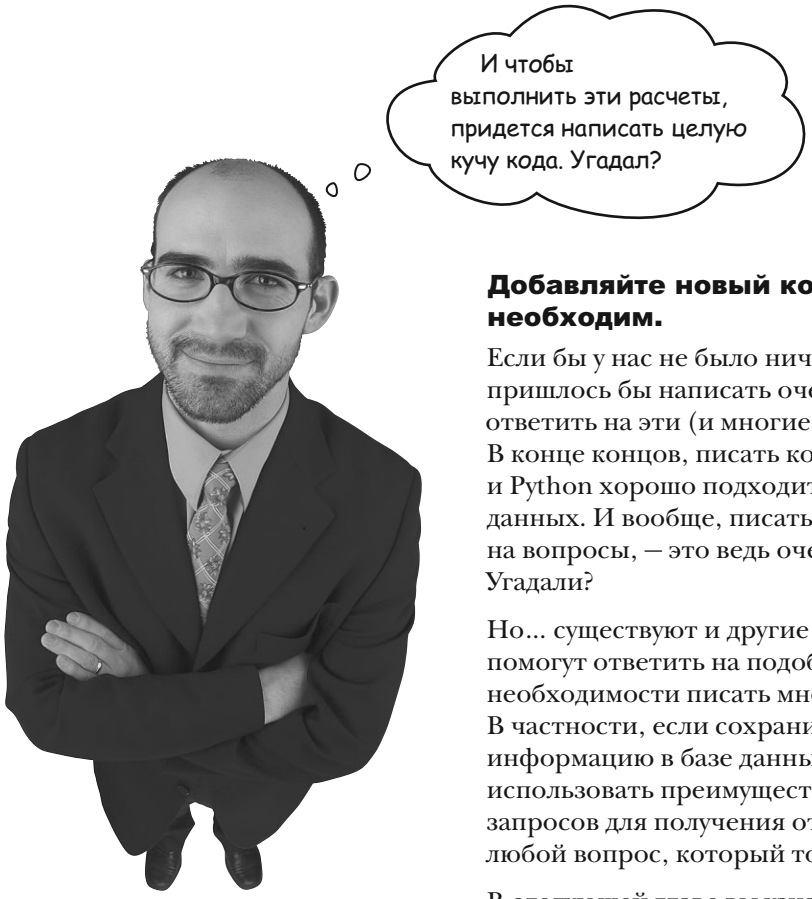
Ln: 2 Col: 0

Задаем вопросы о данных

Возможности веб-приложения заметно расширились, но можем ли мы теперь ответить на вопросы, с которых началась глава? *Сколько запросов было обработано? Какой список букв встречается наиболее часто? С каких IP-адресов приходили запросы? Какой браузер использовался чаще?*

Взглянув на страницу с URL */vieslog*, можно ответить, откуда пришли запросы (колонка **Remote_addr**) и какой браузер использовался (колонка **User_agent**). Но определить, какой браузер посетители сайта используют чаще всего, не так-то просто. Обычного просмотра данных из журнала недостаточно; нужно выполнить дополнительные вычисления.

Ответить на первые два вопроса тоже нелегко. И также необходимы дополнительные вычисления.



Добавляйте новый код, только когда он необходим.

Если бы у нас не было ничего, кроме Python, нам пришлось бы написать очень много кода, чтобы ответить на эти (и многие другие) вопросы. В конце концов, писать код на Python — весело, и Python хорошо подходит для обработки данных. И вообще, писать код, чтобы ответить на вопросы, — это ведь очевидное решение. Угадали?

Но... существуют и другие технологии, которые помогут ответить на подобные вопросы без необходимости писать много кода на Python. В частности, если сохранить журналируемую информацию в базе данных, мы сможем использовать преимущество мощной технологии запросов для получения ответов на практически любой вопрос, который только возникнет.

В следующей главе вы узнаете, как сделать, чтобы веб-приложение хранило информацию в базе данных, а не в текстовом файле.

Код из главы 6

Запомните: оба фрагмента делают одно и то же, но программисты на Python предпочитают этот код (с инструкцией «with») этому.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

```
tasks = open('todos.txt')
for chore in tasks:
    print(chore, end='')
tasks.close()
```

Код, который мы добавили в веб-приложение для поддержки журналирования веб-запросов в текстовом файле.

```
       ...

def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')

       ...


@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)

       ...
```

Мы показали здесь не весь код «vsearch4web.py», а только новые фрагменты. (Целиком программа приведена двумя страницами ранее.)

7 Использование базы данных

Используем DB-API в Python

Хранить информацию в реляционной базе данных очень удобно.

В этой главе вы узнаете, как организовать взаимодействие с популярной базой данных (БД) **MySQL**, используя универсальный прикладной программный интерфейс, который называется **DB-API**. Интерфейс DB-API (входящий в состав стандартной библиотеки Python) позволяет писать код, не зависящий от конкретной базы данных… если база данных понимает SQL. Хотя мы будем использовать MySQL, ничто не помешает вам использовать код DB-API с вашей любимой реляционной базой данных, какой бы она ни была. Давайте посмотрим, как пользоваться реляционной базой данных в Python. В этой главе не так много нового в плане изучения Python, но использование Python для общения с БД — **это важно**, поэтому стоит поучиться.

Включаем поддержку баз данных в веб-приложении

План на главу: изучить все необходимое, чтобы вы смогли изменить свое веб-приложение и организовать хранение журналируемых сведений в базе данных, а не в текстовом файле, как было в главе 6. После этого вы сможете ответить на вопросы, заданные в прошлой главе: *Сколько запросов было обработано? Какой список букв встречается чаще всего? С каких IP-адресов приходили запросы? Какой браузер использовался чаще?*

Вначале нужно решить, какой *системой управления базами данных* (СУБД) мы будем пользоваться. Выбор большой, можно потратить десятки страниц, обсуждая достоинства и недостатки каждой из технологий. Но мы не станем этого делать, а сразу выберем популярную базу данных *MySQL*.

Выбрав MySQL, на следующих десяти страницах мы обсудим четыре важные задачи.

1. **Установка сервера MySQL.**
2. **Установка драйвера MySQL для Python.**
3. **Создание базы данных и таблиц для нашего веб-приложения.**
4. **Программирование операций с базой данных и таблицами.**

Решив эти четыре задачи, мы сможем изменить код `vsearch4web.py` и организовать хранение журналируемых сведений в *MySQL*, а не в текстовом файле. Затем используем SQL, чтобы получить ответы на наши вопросы.

Это не Глупые вопросы

В: Мы обязательно должны использовать здесь MySQL?

О: Если вы хотите выполнять примеры в этой главе, то ответ — да.

В: Можно мне использовать MariaDB вместо MySQL?

О: Да. Поскольку MariaDB — клон MySQL, мы не возражаем против использования MariaDB в качестве базы данных вместо «официальной» MySQL (на самом деле MariaDB — самая популярная база у разработчиков лаборатории Head First).

В: А как же PostgreSQL? Ее можно использовать?

О: Эммм… да, но с оговоркой: если у вас есть опыт использования PostgreSQL (или другой базы данных SQL), вы можете попробовать заменить ею MySQL. Но имейте в виду, что в этой главе нет никаких указаний в отношении использования PostgreSQL (или других СУБД). Вам придется экспериментировать самостоятельно, если что-то в наших примерах с MySQL не будет работать точно так же с вашей СУБД. Есть еще встроенная однопользовательская СУБД **SQLite**, которая поставляется вместе с Python и позволяет работать с SQL *без* необходимости запускать отдельный сервер. Но выбор СУБД зависит от того, что вы собираетесь сделать.

Задача 1. Установка сервера MySQL

- ☐ Установить MySQL на компьютер.
- ☐ Установить драйвер MySQL для Python.
- ☐ Создать базу данных и таблицы.
- ☐ Написать код для чтения/записи данных.

По ходу работы мы будем отмечать каждую завершенную задачу.

Если на вашем компьютере установлен сервер MySQL, вы можете сразу перейти к задаче 2.

Процесс установки MySQL зависит от операционной системы. К счастью, разработчики MySQL (и родственной ей MariaDB) постарались на совесть, чтобы максимально упростить процесс установки.

Если вы используете *Linux*, вам несложно будет найти пакет `mysql-server` (или `mariadb-server`) в репозитории. С помощью диспетчера пакетов (`apt`, `aptitude`, `rpm`, `yum` или другого) установите MySQL как обычный пакет.

Если вы используете *Mac OS X*, советуем установить диспетчер пакетов `Homebrew` (вы найдете его по ссылке *http://brew.sh*), а затем с его помощью установить MariaDB, потому что, судя по нашему опыту, такая комбинация работает лучше.

Во всех других системах (включая различные версии *Windows*) мы советуем установить версию **Community Edition** сервера MySQL, доступную здесь:

http://dev.mysql.com/downloads/mysql/

Или, если вы хотите использовать MariaDB, здесь:

https://mariadb.org/download/

Обязательно прочтите документацию по установке выбранной версии сервера, прежде чем перейти к загрузке и установке.

Не беспокойтесь, если все это ново для вас.

Работая над этой главой, мы не предполагали, что вы знаете MySQL. Мы расскажем все, что необходимо, чтобы вы смогли опробовать примеры (даже если вы никогда раньше не использовали MySQL).

Если хотите немного подучиться, рекомендуем отличную книгу Линн Бейли «Изучаем SQL» (*Head First SQL*) как прекрасный ликбез.

Введение в Python DB-API

- [x] Установить MySQL на компьютер.
- [ ] Установить драйвер MySQL для Python.
- [ ] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

Установив сервер баз данных, оставим его на время в покое и добавим поддержку MySQL в Python.

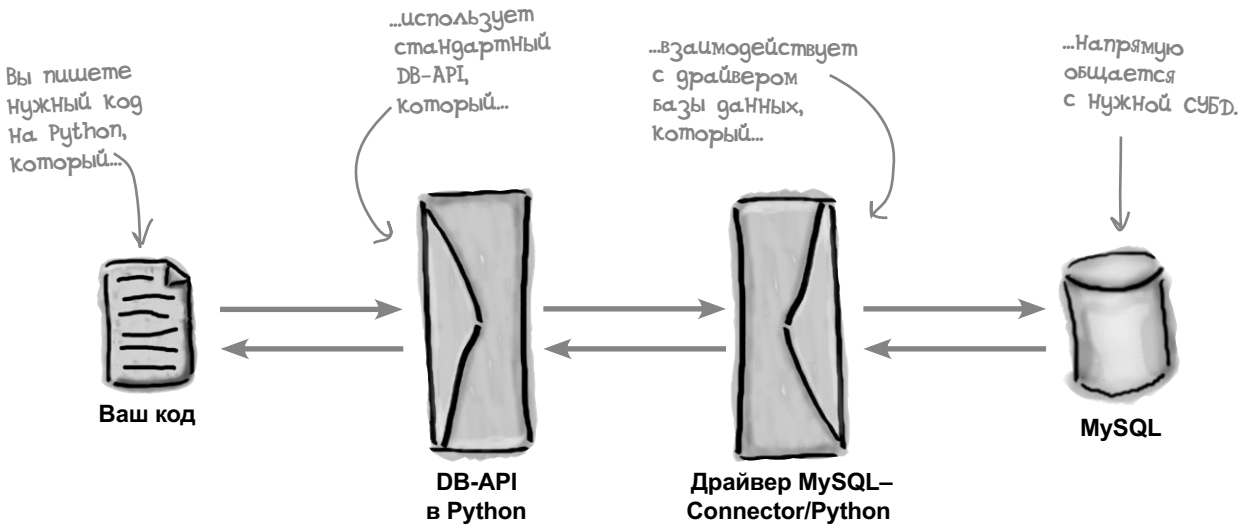
Прямо из коробки интерпретатор Python может работать с базами данных, но не с MySQL. У нас есть стандартный программный интерфейс для работы с базами данных SQL, известный как *DB-API*. Но отсутствует **драйвер**, позволяющий соединить DB-API с конкретной СУБД, которую вы собираетесь использовать.

В соответствии с соглашениями, для взаимодействий с любой базой данных из Python программисты используют DB-API, потому что драйвер избавляет программистов от необходимости понимать тонкости взаимодействия с фактическим программным интерфейсом базы данных, а DB-API образует дополнительный уровень абстракции. Используя DB-API, вы можете при необходимости заменить СУБД, и вам не придется переписывать существующий код.

Мы еще поговорим о DB-API в этой главе. А пока взгляните на диаграмму, изображающую порядок использования DB-API в Python.

Для умников

DB-API для Python описан в PEP 0247. Однако нет необходимости сразу же читать эту документацию, потому что в первую очередь это свод требований для разработчиков драйверов баз данных (а не пошаговое руководство).



Некоторые программисты, взглянув на эту диаграмму, приходят к выводу, что использование DB-API в Python должно быть крайне неэффективным. В конце концов, тут *два* уровня абстракции между прикладным кодом и СУБД. Однако DB-API позволяет переключаться между базами данных без необходимости переписывать код, которая неизбежно возникает при *непосредственном* обращении к базе данных. Если еще учесть, что нет двух одинаковых диалектов SQL, становится очевидным, что использование DB-API помогает достичь более высокого уровня абстракции.

Задача 2. Установка драйвера базы данных MySQL для Python

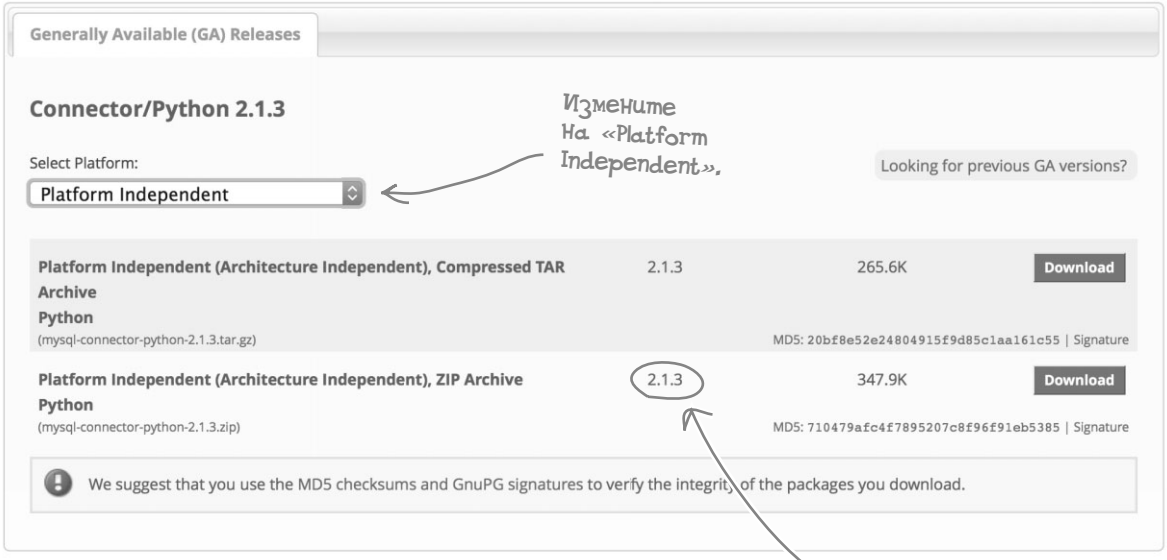
- [x] Установить MySQL на компьютер.
- [ ] Установить драйвер MySQL для Python.
- [ ] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

Любой может написать драйвер базы данных (и многие этим занимаются), но обычно производитель базы данных предоставляет *официальный драйвер* для каждого поддерживаемого языка программирования. Компания *Oracle*, производитель MySQL, предоставляет драйвер *MySQL-Connector/Python*, и мы будем использовать его в этой главе. Есть только одна проблема: *MySQL-Connector/Python* нельзя установить при помощи `pip`.

Означает ли это, что нам очень не повезло с *MySQL-Connector/Python с Python*? Совсем нет. Тот факт, что сторонний модуль не поддерживает работу с механизмом `pip`, редко оказывается серьезным препятствием. Чтобы установить модуль вручную, придется проделать дополнительную работу (по сравнению с использованием `pip`), но совсем немного.

Там и поступим. Установим драйвер *MySQL-Connector/Python* вручную (доступны и *другие* драйверы, например *PyMySQL*; однако мы предпочитаем *MySQL-Connector/Python*, потому что это официально поддерживаемый драйвер от создателей MySQL).

Чтобы загрузить *MySQL-Connector/Python*, зайдите на страницу *https://dev.mysql.com/downloads/connector/python/*. Вполне возможно, что страница автоматически выберет вашу операционную систему в раскрывающемся списке *Select Platform* (*Выбор платформы*). Проигнорируйте это предложение и выберите пункт *Platform Independent* (*Независимый от платформы*), как показано здесь.



Затем щелкните на любой из кнопок *Download* (*Загрузить*). Обычно пользователи *Windows* предпочитают ZIP-файлы, а пользователи *Linux* и *Mac OS X* выбирают GZ. Сохраните загруженный файл на компьютере, затем дважды щелкните на нем, чтобы распаковать в той же папке, куда он был загружен.

Установка MySQL-Connector/Python

- [x] Установить MySQL на компьютер.
- [ ] Установить драйвер MySQL для Python.
- [ ] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

Загрузив и распаковав драйвер, откройте окно терминала в только что созданной папке (если вы работаете в *Windows*, откройте терминал с правами администратора, выбрав в контекстном меню пункт *Run as Administrator* (*Запуск от имени администратора*).

Будем считать, что в результате распаковывания драйвера внутри папки `Downloads` была создана папка `mysql-connector-python-2.1.3`. Чтобы установить драйвер в *Windows*, выполните эту команду в командной строке, в папке `mysql-connector-python-2.1.3`.

```
py  -3  setup.py  install
```

В *Linux* и *Mac OS X* выполните в командной строке такую команду:

```
sudo  -H  python3  setup.py  install
```

Независимо от операционной системы, в результате выполнения данных команд на экране появится целый набор различных сообщений, например:

У вас могут быть другие пути. Не беспокойтесь, если так и есть.

```
running install
Not Installing C Extension
running build
running build_py
running install_lib
running install_egg_info
Removing /Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
mysql_connector_python-2.1.3-py3.5.egg-info
Writing /Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
mysql_connector_python-2.1.3-py3.5.egg-info
```

Когда модуль устанавливает диспетчер пакетов `pip`, он действует примерно так же, но скрывает подобные сообщения от вас. Сообщения, которые вы видите здесь, указывают, что процесс установки протекает гладко. Если что-то пошло не так, сообщение об ошибке будет содержать достаточно информации, чтобы устранить проблему. Если все прошло нормально, появление таких сообщений подтвердит, что *MySQL-Connector/Python* готов к использованию.

Это не Глупые вопросы

В: Должно ли меня обеспокоить сообщение «Not Installing C Extension» (Расширение на C не установлено)?

О: Нет. Иногда сторонние модули содержат встроенный код на C, помогающий ускорить вычисления. Однако не все операционные системы поставляются с предустановленным компилятором языка C, поэтому вы должны отдельно указать, что вместе с модулем необходимо устанавливать расширения на C (если вы решили, что вам это нужно). В противном случае вместо расширения на C будет установлен модуль на языке Python (потенциально более медленный). Это позволит модулю работать на любой платформе, вне зависимости от наличия компилятора C. Когда сторонний модуль использует только код на Python, его называют модулем, написанным на «чистом Python». В нашем примере устанавливается версия *MySQL–Connector/Python*, написанная на чистом Python.

Задача 3. Создание базы данных и таблиц для веб-приложения

- [x] Установить MySQL на компьютер.
- [x] Установить драйвер MySQL для Python.
- [ ] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

Итак, вы установили сервер баз данных MySQL и драйвер *MySQL-Connector/Python*. Самое время решить третью задачу — создать базу данных и таблицы для нашего веб-приложения.

Для этого будем взаимодействовать с сервером MySQL с помощью специального инструмента командной строки — небольшой утилиты, запускаемой в окне терминала. Этот инструмент известен как *консоль* MySQL. Следующая команда запустит консоль и осуществит вход от имени администратора базы данных MySQL (с идентификатором пользователя `root`).

```
mysql  -u  root  -p
```

Если вы назначили пароль администратора во время установки сервера MySQL, введите его после нажатия клавиши *Enter*. Если пароль не назначался, просто нажмите *Enter* дважды. В любом случае вы попадете в **приглашение к вводу**, которое выглядит так (слева — для MySQL, справа — для MariaDB).

```
mysql>                          MariaDB [None]>
```

Любые команды, введенные в приглашении, передаются для выполнения серверу MySQL. Начнем с создания базы данных для веб-приложения. Мы собирались использовать базу данных для хранения журналируемых сведений, поэтому имя базы данных должно отражать ее назначение. Назовем ее `vsearchlogDB`. Следующая команда создает базу данных.

```
mysql> create database vsearchlogDB;
```

Не забывайте добавлять точку с запятой в конце каждой команды в консоли MySQL.

В консоли появится (малопонятное) сообщение: `Query OK, 1 row affected (0.00 sec)`. Таким способом консоль сообщает, что все идет отлично.

Теперь создадим учетную запись для базы данных, чтобы при работе с MySQL наше веб-приложение использовало ее вместо `root` (использование `root` считается плохой практикой). Следующая команда создает новую учетную запись пользователя MySQL с именем `vsearch` и паролем «vsearchpasswd», предоставляющую все права для работы с базой данных `vsearchlogDB`.

```
mysql> grant all on vsearchlogDB.* to 'vsearch' identified by 'vsearchpasswd';
```

Если хотите, можете использовать другой пароль. Главное, не забудьте использовать его в последующих примерах.

Должно появиться похожее сообщение `Query OK`, свидетельствующее, что учетная запись создана. Теперь выйдем из консоли с помощью этой команды.

```
mysql> quit
```

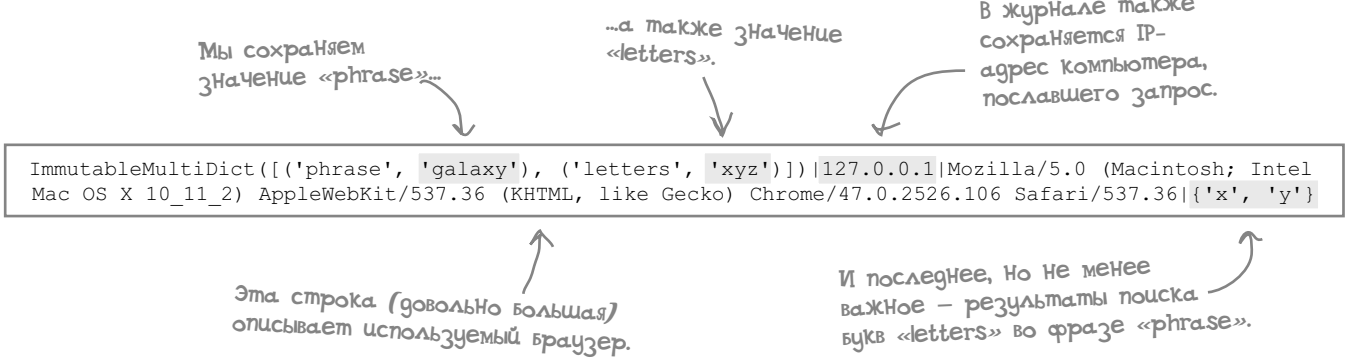
Вы увидите дружелюбное сообщение `Bye` в консоли, а затем вернетесь в командную строку своей операционной системы.

Выбираем структуру для журналируемых данных

- [x] Установить MySQL на компьютер.
- [x] Установить драйвер MySQL для Python.
- [ ] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

После создания базы данных нужно добавить в нее таблицы (необходимые приложению). Нам достаточно одной таблицы для сохранения информации об отдельных веб-запросах.

Вспомните, как мы сохраняли данные в текстовый файл в предыдущей главе: каждая строка в файле `vsearch.log` соответствовала определенному формату.



Наконец, нам нужна таблица с пятью полями: для фразы, букв, IP-адреса, строки, описывающей браузер, и результатов поиска. Давайте добавим еще два поля: уникальный идентификатор запроса и время его записи в журнал. Эти два столбца встречаются так часто, что в MySQL предусмотрен простой способ добавления таких данных в каждую запись, и мы покажем его в конце страницы.

Нужную структуру таблицы можно задать прямо в консоли. Но сначала войдем под именем нашего нового пользователя `vsearch`, выполнив следующую команду (не забудьте ввести правильный пароль после нажатия клавиши *Enter*).

```
mysql -u vsearch -p vsearchlogDB
```

Как вы помните, мы установили пароль «vsearchpasswd».

Следующая инструкция на языке SQL создает таблицу (с именем `log`). Обратите внимание: символы `->` не являются частью инструкции, они добавляются автоматически, когда консоль ожидает ввода дополнительных параметров (в многострочных командах SQL). Инструкция должна завершаться символом точки с запятой и запускается нажатием клавиши *Enter*.

```
mysql> create table log (
    -> id int auto_increment primary key,
    -> ts timestamp default current_timestamp,
    -> phrase varchar(128) not null,
    -> letters varchar(32) not null,
    -> ip varchar(16) not null,
    -> browser_string varchar(256) not null,
    -> results varchar(64) not null );
```

Стрелка означает продолжение ввода команды.

MySQL автоматически записывает данные в эти поля.

Поля хранят сведения о каждом запросе (из формы).

Убедимся, что таблица готова к использованию

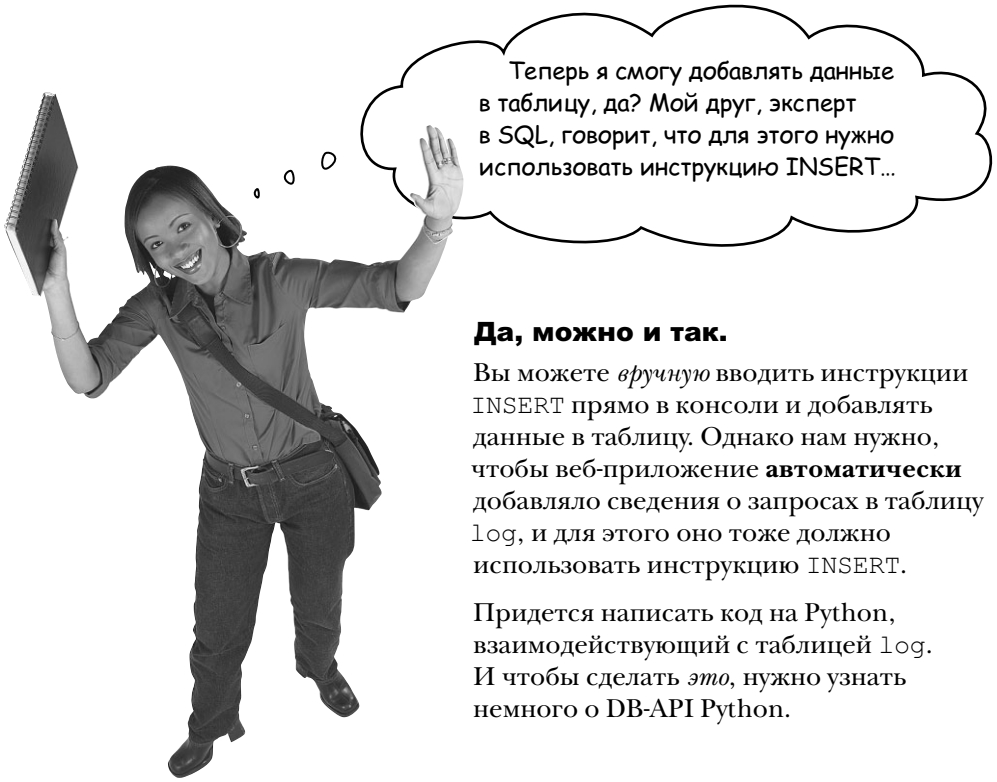
- [x] Установить MySQL на компьютер.
- [x] Установить драйвер MySQL для Python.
- [x] Создать базу данных и таблицы.
- [ ] Написать код для чтения/записи данных.

Создав таблицу, мы выполнили третью задачу.

Давайте убедимся, что таблица действительно создана и имеет нужную структуру. Не выходя из консоли MySQL, выполните команду ***describe log***.

```
mysql> describe log;
+----------------+-------------+------+-----+-------------------+----------------+
| Field          | Type        | Null | Key | Default           | Extra          |
+----------------+-------------+------+-----+-------------------+----------------+
| id             | int(11)     | NO   | PRI | NULL              | auto_increment |
| ts             | timestamp   | NO   |     | CURRENT_TIMESTAMP |                |
| phrase         | varchar(128)| NO   |     | NULL              |                |
| letters        | varchar(32) | NO   |     | NULL              |                |
| ip             | varchar(16) | NO   |     | NULL              |                |
| browser_string | varchar(256)| NO   |     | NULL              |                |
| results        | varchar(64) | NO   |     | NULL              |                |
+----------------+-------------+------+-----+-------------------+----------------+
```

Вот доказательство того, что таблица `log` существует и ее структура соответствует требованиям приложения. Наберите ***quit*** и выйдите из консоли (если вы этого еще не сделали).



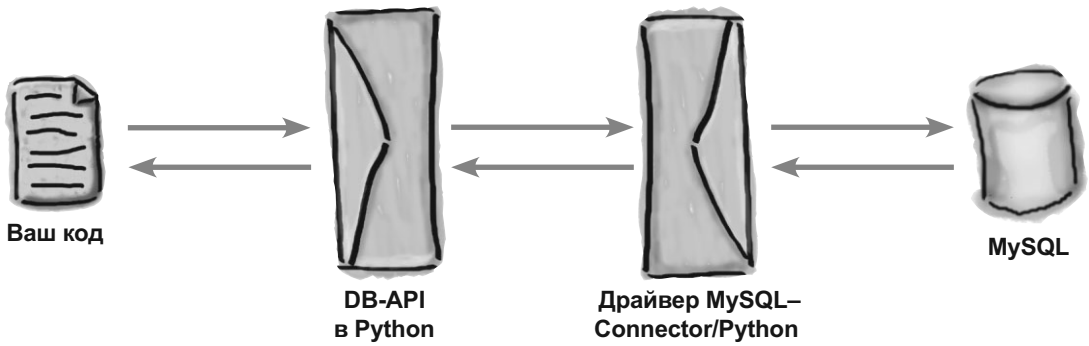
Да, можно и так.

Вы можете *вручную* вводить инструкции `INSERT` прямо в консоли и добавлять данные в таблицу. Однако нам нужно, чтобы веб-приложение **автоматически** добавляло сведения о запросах в таблицу `log`, и для этого оно тоже должно использовать инструкцию `INSERT`.

Придется написать код на Python, взаимодействующий с таблицей `log`. И чтобы сделать *это*, нужно узнать немного о DB-API Python.

DB-API под лупой, 1 из 3

Вспомните диаграмму, приведенную в начале главы, которая изображает порядок использования DB-API, драйвера СУБД и самой СУБД в Python.



Используя DB-API, вы легко сможете заменить комбинацию из драйвера/базы данных, почти не меняя свой код на Python, хотя и ограничив себя только возможностями DB-API.

Посмотрим, что включает в себя этот важный стандарт программирования на Python. Далее перечислены шесть основных шагов.

DB-API, шаг 1. Задаем характеристики соединения с базой данных

Для соединения с MySQL необходимо определить четыре параметра: IP-адрес или сетевое имя компьютера, на котором выполняется сервер MySQL (1); имя пользователя (2); пароль, соответствующий этому пользователю (3); имя базы данных (4).

Драйвер *MySQL-Connector/Python* позволяет передавать эти параметры в словаре Python, упрощая их использование и обращение с ними. Сейчас мы создадим словарь в интерактивной оболочке >>>. Выполните этот пример. Вот словарь (с именем `dbconfig`), в котором находятся четыре необходимых для соединения ключа с соответствующими значениями.

1. Наш сервер запущен на локальном компьютере, поэтому для ключа «host» мы использовали локальный IP-адрес.

```
>>> dbconfig = { 'host': '127.0.0.1',
              'user': 'vsearch',
              'password': 'vsearchpasswd',
              'database': 'vsearchlogDB', }
```

2. Имя учетной записи «vsearch», созданной ранее в главе, сохраняется с ключом «user».

3. Ключу «password» присваивается пароль, который мы использовали для доступа к базе данных под именем нашей учетной записи.

4. Имя базы данных — «vsearchlogDB» — присвоено ключу «database».

DB-API, шаг 2. Импорт драйвера базы данных

После определения параметров соединения можно импортировать драйвер базы данных.

```
>>> import mysql.connector
```

Импорт драйвера используемой базы данных.

Теперь драйвер MySQL становится доступным для DB-API.

DB-API, шаг 3. Установка соединения с сервером

Подключимся к серверу с помощью функции `connect` из DB-API и сохраним ссылку на соединение в переменной `conn`. Вот вызов функции `connect`, которая устанавливает соединение с сервером баз данных MySQL (и создает `conn`):

```
>>> conn = mysql.connector.connect(**dbconfig)
```

Вызов устанавливает соединение.

Передача словаря с параметрами соединения.

Обратите внимание на странные звездочки `**`, предшествующие единственному аргументу функции `connect`. (Если вы программист на C/C++, **не надо** читать `**` как «указатель на указатель», потому что в Python нет указателей.) Звездочки `**` сообщают функции `connect`, что все аргументы передаются ей в единственном словаре (в данном случае — в только что созданном словаре `dbconfig`). Встретив `**`, функция `connect` развернет словарь в четыре отдельных аргумента, которые потом будут использованы для подключения к серверу. (Вы еще не раз встретите звездочки `**` в следующих главах, а пока просто используйте этот код.)

DB-API, шаг 4. Открытие курсора

Чтобы отправить базе данных SQL-команды (через только что открытое соединение) и получить результаты, нужен *курсор*. Курсор можно считать эквивалентом *дескриптора файла* (который позволяет взаимодействовать с файлом на диске после его открытия), но только для базы данных.

Создать курсор несложно. Для этого достаточно вызвать метод `cursor` любого объекта соединения. Мы сохраним ссылку на созданный курсор в переменной (которую, не мудрствуя лукаво, назвали `cursor`).

```
>>> cursor = conn.cursor()
```

Создание курсора для отправки команд на сервер и получения результатов.

Теперь мы можем отправлять команды SQL на сервер и — надеемся — получать результаты.

Остановимся на минуту и обсудим выполненные действия. Мы определили параметры соединения с базой данных, импортировали модуль драйвера, создали объекты соединение и курсор. Эти этапы не зависят от базы данных и выполняются перед любыми взаимодействиями с MySQL (изменяются только параметры подключения). Учитывайте это, когда будете работать с данными через курсор.

DB-API под лупой, 2 из 3

Курсор создан и присвоен переменной, можно приступать к работе с базой данных при помощи языка запросов SQL.

DB-API, шаг 5. Работаем с SQL!

Переменная `cursor` позволяет передавать SQL-запросы в СУБД MySQL и получать результаты их обработки.

Обычно программисты на Python из лаборатории *Head First* оформляют код SQL, который намереваются отправить серверу баз данных, в виде строки в тройных кавычках, а затем присваивают эту строку переменной `_SQL`. Запросы SQL часто занимают несколько строк, поэтому для их записи используются строки в тройных кавычках, которые являются исключением из общего правила Python «конец строки — конец выражения». Использование `_SQL` в качестве имени переменной — лишь соглашение по именованию констант, принятое программистами из лаборатории *Head First*, но вы можете использовать любое другое имя (оно необязательно должно состоять только из букв верхнего регистра или начинаться с подчеркивания).

Попросим MySQL сообщить имена всех таблиц в базе данных. Для этого присвойте переменной `_SQL` запрос `show tables`, а затем вызовите функцию `cursor.execute`, передав ей `_SQL` в качестве аргумента.

Присвоить запрос SQL переменной.

```
>>> _SQL = """show tables"""
>>> cursor.execute(_SQL)
```

Передать запрос в переменной «_SQL» серверу MySQL для выполнения.

Введите команду `cursor.execute` в интерактивной оболочке >>>, и SQL-запрос отправится на ваш сервер MySQL, который обработает запрос (если он содержит допустимый и корректный код на языке SQL). Однако результаты сами не появятся; их нужно запросить.

Запросить результаты можно, используя один из трех методов курсора:

- `cursor.fetchone` возвращает записи из результата **по одной**;
- `cursor.fetchmany` возвращает заданное **количество** записей;
- `cursor.fetchall` возвращает **все** записи.

Используем метод `cursor.fetchall`, чтобы получить сразу все записи из результатов обработки нашего запроса, и присвоим их переменной `res`, содержимое которой выведем в консоли >>>.

Получить все записи, полученные от MySQL.

```
>>> res = cursor.fetchall()
>>> res
[('log',)]
```

Отобразить результаты.

Содержимое `res` немного удивляет, не так ли? Возможно, вы ожидали увидеть одно слово, потому что раньше мы говорили, что наша база данных (`vsearchlogDB`) содержит единственную таблицу `log`. Однако метод `cursor.fetchall` всегда возвращает *список кортежей*, даже если получена всего одна запись данных (как в нашем случае). Давайте рассмотрим еще один пример, где MySQL возвращает больше данных.

Следующий запрос — `describe log` — запрашивает информацию о таблице `log` в базе данных. Ниже эта информация выводится *дважды*: один раз в простом, неформатированном виде (и выглядит немного запутанной) и затем в нескольких строках. Напомним, что `cursor.fetchall` возвращает список кортежей.

Вот `cursor.fetchall` в действии снова.

```
>>> _SQL = """describe log"""
>>> cursor.execute(_SQL)
>>> res = cursor.fetchall()
>>> res
[('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'), ('ts',
'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', ''), ('phrase',
'varchar(128)', 'NO', '', None, ''), ('letters', 'varchar(32)',
'NO', '', None, ''), ('ip', 'varchar(16)', 'NO', '', None, ''),
('browser_string', 'varchar(256)', 'NO', '', None, ''), ('results',
'varchar(64)', 'NO', '', None, '')]
```

Взять SQL запрос...

...отправить его на сервер...

...и получить результаты.

Выглядит немного запутанно, но это список кортежей.

```
>>> for row in res:
      print(row)
```

Каждую строку результата...

...отобразить в отдельной строке.

Каждый кортеж из списка кортежей теперь выводится в отдельной строке.

```
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('ts', 'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', '')
('phrase', 'varchar(128)', 'NO', '', None, '')
('letters', 'varchar(32)', 'NO', '', None, '')
('ip', 'varchar(16)', 'NO', '', None, '')
('browser_string', 'varchar(256)', 'NO', '', None, '')
('results', 'varchar(64)', 'NO', '', None, '')
```

Построчный вывод может показаться не лучше неформатированного, но сравните его с выводом в консоли MySQL, на рисунке ниже. Это те же данные (что и выше), только у нас они хранятся в переменной `res`.

```
mysql> describe log;
+----------------+--------------+------+-----+-------------------+----------------+
| Field          | Type         | Null | Key | Default           | Extra          |
+----------------+--------------+------+-----+-------------------+----------------+
| id             | int(11)      | NO   | PRI | NULL              | auto_increment |
| ts             | timestamp    | NO   |     | CURRENT_TIMESTAMP |                |
| phrase         | varchar(128) | NO   |     | NULL              |                |
| letters        | varchar(32)  | NO   |     | NULL              |                |
| ip             | varchar(16)  | NO   |     | NULL              |                |
| browser_string | varchar(256) | NO   |     | NULL              |                |
| results        | varchar(64)  | NO   |     | NULL              |                |
+----------------+--------------+------+-----+-------------------+----------------+
```

Посмотрите внимательно — это те же данные.

DB-API под лупой, 3 из 3

Используем запрос `insert` и добавим какие-нибудь данные в таблицу `log`.

Весьма соблазнительно присвоить переменной `_SQL` строку запроса, показанную ниже (мы записали ее в нескольких строках), а затем, вызвав `cursor.execute`, передать запрос на сервер.

```
>>> _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              ('hitch-hiker', 'aeiou', '127.0.0.1', 'Firefox', "{'e', 'i'}")"""
>>> cursor.execute(_SQL)
```

Не поймите нас неправильно, все показанное выше работает. Однако данные, *жестко закодированные* таким способом, редко бывают именно тем, что вам нужно, потому что данные для сохранения в таблице, скорее всего, будут меняться при каждом вызове `insert`. Вспомните: мы планируем сохранять в таблице `log` данные о веб-запросах, а это означает, что они *будут* меняться в каждом запросе, поэтому явно определять их в теле запроса — путь к неприятностям.

Чтобы избавить от необходимости жестко вбивать данные (как показано выше), DB-API позволяет оставить в строке запроса место для данных с помощью меток-заполнителей, которые будут замещаться фактическими значениями при вызове `cursor.execute`. Такой способ позволяет многократно использовать строку запроса с различными данными, передавая значения аргументов в запрос непосредственно перед его выполнением. Метки-заполнители в нашем запросе представляют строковые значения и обозначаются как `%s` в коде внизу.

Сравните эти команды с командами выше.

При составлении запроса используйте метки-заполнители вместо фактических значений.

```
>>> _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
>>> cursor.execute(_SQL, ('hitch-hiker', 'xyz', '127.0.0.1', 'Safari', 'set()'))
```

Необходимо отметить следующие два момента. Во-первых, вместо жестко заданных данных мы использовали в запросе SQL метки-заполнители `%s`, которые сообщают DB-API, что есть дополнительные строковые значения, которые нужно вставить в запрос перед выполнением. Как видите, в запросе присутствует пять меток-заполнителей `%s`, поэтому, во-вторых, нужно отметить, что в вызов `cursor.execute` необходимо передать пять дополнительных параметров. Единственная проблема в том, что `cursor.execute` принимает *самое большее* два аргумента.

Как такое возможно?

Последняя строка кода в примере выше ясно показывает, что `cursor.execute` принимает *пять* значений данных (и без всяких жалоб), так что происходит?

Взгляните внимательнее на этот код. Обратите внимание на скобки вокруг значений данных. Они автоматически превращают пять значений в один кортеж (хранящий эти отдельные значения). В итоге в `cursor.execute` передаются два аргумента: запрос с метками-заполнителями, а также кортеж с данными.

Итак, когда этот код выполнится, данные будут вставлены в таблицу `log`, так ведь? Ну... не совсем.

Когда для отправки данных в базу данных (с помощью запроса `insert`) используется `cursor.execute`, данные могут не сохраниться в базе данных немедленно. Это происходит потому, что запись в базу данных — операция **дорогостоящая** (с точки зрения процесса обработки), поэтому многие базы данных кэшируют запросы `insert`, а потом, чуть погодя, выполняют их все сразу. Поэтому иногда, когда вы думаете, что ваши данные уже в базе данных, их там еще нет. И это может привести к проблемам.

Например, если послать запрос `insert` для вставки данных в таблицу, а затем сразу выполнить запрос `select`, чтобы прочитать их, данные могут оказаться недоступными, поскольку все еще ожидают записи в кэше. Если такое случится, значит вам не повезло и `select` не вернет ваши данные. В конечном итоге данные будут записаны — они не потеряются, но кэширование может спутать ваши замыслы.

Однако если вы готовы пожертвовать некоторой долей производительности ради немедленной записи данных, можете явно потребовать от базы данных сохранить все кэшированные данные в таблицу, вызвав метод `conn.commit`. Сейчас мы так и поступим, чтобы убедиться, что наши два запроса `insert` с предыдущей страницы успешно записали данные в таблицу `log`. После записи данные можно читать, выполняя запросы `select`:

«Заставим» записать кэшированные данные в таблицу.

```
>>> conn.commit()
>>> _SQL = """select * from log"""
>>> cursor.execute(_SQL)
>>> for row in cursor.fetchall():
      print(row)
```

Получим только что записанные данные.

Это идентификатор «id», автоматически присвоенный MySQL этой записи...

```
(1, datetime.datetime(2016, 3, ..., "{'e', 'i'}")
(2, datetime.datetime(2016, 3, ..., 'set()')
```

...а вот что оказалось на месте «ts» (timestamp).

Мы немного сократили вывод, чтобы уместить по ширине страницы.

Из примера выше видно, что MySQL автоматически определяет корректные значения для полей `id` и `ts` во время вставки данных. Как и раньше, сервер возвращает данные в виде списка кортежей. Но теперь мы не стали сохранять результат вызова `cursor.fetchall` в переменной, а использовали вызов `cursor.fetchall` прямо в цикле `for`. Не забывайте: кортеж — это неизменяемый список, он поддерживает синтаксис доступа к данным с квадратными скобками. Это означает, что можно использовать индексы для доступа к отдельным элементам данных в переменной `row`, которая является переменной цикла. Например, `row[2]` вернет фразу, `row[3]` — искомые буквы, а `row[-1]` — результаты поиска.

DB-API, шаг 6. Закрываем курсор и соединение

После сохранения данных в таблице нужно убрать за собой, закрыв курсор и соединение:

```
>>> cursor.close()
True
>>> conn.close()
```

Убирать за собой — всегда хорошая идея.

Обратите внимание, что курсор подтверждает успешное закрытие, вернув `True`, а операция закрытия соединения ничего не возвращает. Закрывать курсор и соединение, когда они больше не нужны, всегда правильное решение, потому что база данных располагает ограниченным количеством ресурсов. В лаборатории *Head First* программисты всегда держат открытыми курсоры и соединения столько, сколько это нужно, но не дольше.

Задача 4. Программирование операций с базой данных и таблицами

- ☑ Установить MySQL на компьютер.
- ☑ Установить драйвер MySQL для Python.
- ☑ Создать базу данных и таблицы.
- ☑ Написать код для чтения/записи данных.

Наш список задач выполнен!

Мы выполнили шесть *шагов* в разделе «DB-API под лупой», теперь напишем код для взаимодействия с таблицей `log`, и когда мы сделаем это, задача 4 будет завершена: *программирование операций с базой данных и таблицами*.

Взглянем еще раз на код (целиком), который можно использовать:

Определить параметры соединения.

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

import mysql.connector

conn = mysql.connector.connect(**dbconfig)

cursor = conn.cursor()

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""

cursor.execute(_SQL, ('galaxy', 'xyz', '127.0.0.1', 'Opera', "{'x', 'y'}"))

conn.commit()

_SQL = """select * from log"""

cursor.execute(_SQL)

for row in cursor.fetchall():
      print(row)

cursor.close()

conn.close()
```

Импортировать драйвер базы данных.

Присвоить строку запроса переменной (обратите внимание на пять меток-заполнителей для аргументов).

Установить соединение и создать курсор.

Послать запрос серверу, подставив значения для всех аргументов (в кортеже).

Принудительно записать данные в таблицу.

Получить (только что записанные) данные из таблицы, отобразить их строку за строкой.

Убрать за собой по окончании.

Все четыре задачи выполнены, и мы готовы изменить веб-приложение, чтобы оно сохраняло сведения о веб-запросах в базе данных MySQL, а не в файле (как было раньше).

Магнитики для базы данных

Еще раз взгляните на функцию `log_request` из предыдущей главы.

Эта маленькая функция принимает два аргумента: объект веб-запроса и результаты вызова `vsearch`.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

Ваша задача — изменить тело функции так, чтобы она сохраняла информацию в базе данных (а не в текстовом файле). Строка `def` остается без изменений. Решите, какие магнитики нужны из тех, что разбросаны внизу страницы, и расставьте их по местам.

```
def log_request(req: 'flask_request', res: str) -> None:
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

Ну и путаница с этими магнитами! Поможете?

```
conn.close()
```

```
cursor = conn.cursor()
```

```
conn.commit()
```

```
_SQL = """select * from log"""
```

```
import mysql.connector
```

```
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))
```

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

```
_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""
```

```
for row in cursor.fetchall():
    print(row)
```

```
cursor.close()
```

```
cursor.execute(_SQL)
```

```
conn = mysql.connector.connect(**dbconfig)
```

Магнитики для базы данных. Решение

Вы должны были еще раз взглянуть на функцию `log_request` из предыдущей главы.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, file=log, sep='|')
```

Задача — изменить тело функции так, чтобы она сохраняла информацию в базе данных. Строка `def` должна остаться без изменений. Решите, какие магнитики нужны из тех, что разбросаны внизу предыдущей страницы.

```
def log_request(req: 'flask_request', res: str) -> None:
```

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

Определить параметры соединения.

```
import mysql.connector
```

```
conn = mysql.connector.connect(**dbconfig)
```

```
cursor = conn.cursor()
```

Импортировать драйвер, установить соединение и создать курсор.

```
_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""
```

Создать строку с текстом запроса для записи в БД.

```
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))
```

Выполнить запрос.

Новая часть: из строки с описанием браузера (хранящейся в «req.user_agent») извлекается только его название.

```
conn.commit()
```

```
cursor.close()
```

```
conn.close()
```

После записи данных убираем за собой, закрыв курсор и соединение.

А эти магнитики не пригодились.

```
cursor.execute(_SQL)
```

```
_SQL = """select * from log"""
```

```
for row in cursor.fetchall():
    print(row)
```

Пробная поездка

Измените функцию `log_request` в своем файле `vsearch4web.py`, как было показано на предыдущей странице. Сохраните код и запустите обновленную версию веб-приложения в командной строке. Напомним, что в *Windows* для этого нужно выполнить команду.

```
C:\webapps> py -3 vsearch4web.py
```

А в *Linux* или *Mac OS X* используйте такую команду.

```
$ python3 vsearch4web.py
```

Веб-приложение должно начать обслуживать следующий веб-адрес.

http://127.0.0.1:5000/

С помощью веб-браузера выполните несколько запросов и убедитесь, что приложение работает нормально.

Своими действиями мы добились следующего.

- Приложение работает как и прежде: в ответ на каждый запрос возвращается страница результатов.
- Пользователи не догадываются, что теперь результаты запросов сохраняются в таблице базы данных, а не в текстовом файле.

К сожалению, теперь нельзя посмотреть журнал по URL */viewlog*, потому что функция, связанная с этим URL (`view_the_log`), работает только с текстовым файлом `vsearch.log` (а не с базой данных). Скоро мы расскажем, как это исправить.

Теперь подведем итог «Пробной поездки» и с помощью консоли MySQL убедимся, что новая версия `log_request` сохраняет данные в таблице `log`. Откройте еще одно окно терминала и выполняйте команды вместе с нами (обратите внимание: мы переформатировали и сократили вывод, чтобы уместить его по ширине страницы).

Вход в консоль MySQL.

Запрос требует вернуть все данные из таблицы «log» (ваши данные, скорее всего, будут отличаться).

```
File Edit Window Help Checking our log DB
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                  | letters | ip        | browser_string | results              |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker             | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                  | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker             | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

Не забудьте выйти из консоли, когда закончите работу.

Помните, что мы сохраняем только имя браузера.

Хранение данных — только половина дела

Во время «Пробной поездки» на предыдущей странице вы убедились, что новая функция `log_request`, использующая DB-API, действительно сохраняет сведения о веб-запросах в таблице `log`.

Еще раз взгляните на последнюю версию функции `log_request` (включающую строку документации в самом начале).

```
def log_request(req: 'flask_request', res: str) -> None:
    """Журналирует веб-запрос и возвращаемые результаты."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    import mysql.connector

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Опытные программисты на Python могут взглянуть на этот код и неодобрительно хмыкнуть. Почему — вы узнаете через несколько страниц.

Новая функция — совершенно другая

Функция `log_request` получилась намного длиннее прежней версии, работавшей с текстовым файлом, но для работы с базой данных MySQL (которую мы будем использовать, чтобы ответить на вопросы о данных в журнале, заданные в конце предыдущей главы) необходим дополнительный код, поэтому увеличение и усложнение новой версии `log_request` выглядит оправданным.

Теперь вспомним, что в веб-приложении есть другая функция, `view_the_log`, которая получает данные из файла `vsearch.log` и отображает их в приятном для глаза формате на веб-странице. Нам нужно изменить `view_the_log`, чтобы она получала данные из таблицы `log` в базе данных, а не из текстового файла.

Вопрос: как это сделать лучше всего?

Как организовать код для работы с базой данных?

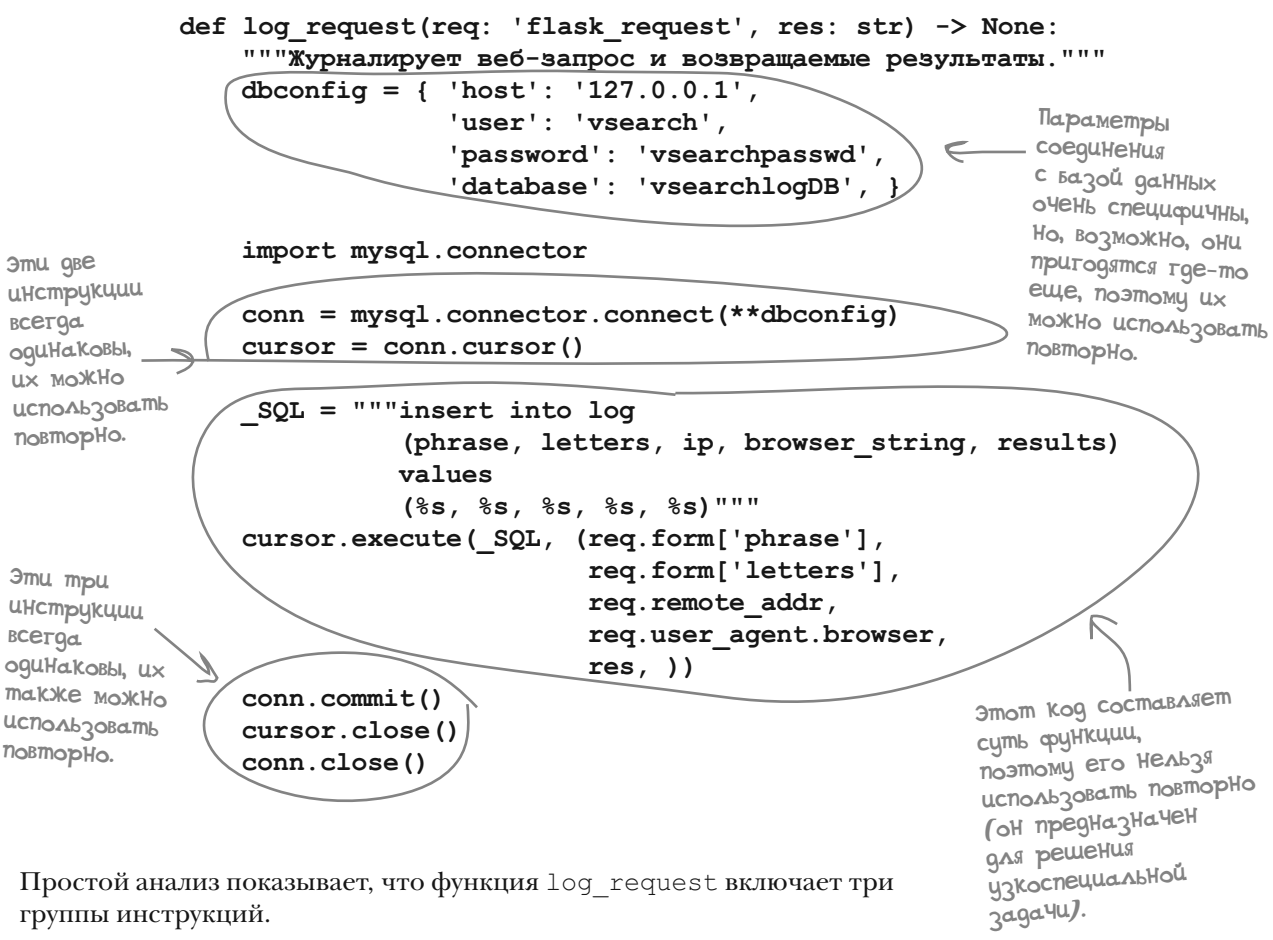
Теперь у вас есть код, сохраняющий информацию о запросах к веб-приложению в MySQL. Используя нечто похожее, мы без особого труда сможем научить функцию `view_the_log` извлекать данные из таблицы `log`. Но как лучше это сделать? Мы спросили трех программистов... и получили три разных ответа.

Все предложения по-своему верны, хотя и выглядят немного подозрительно (особенно первое). Возможно, вас удивит, но в данном случае программисты на Python *вряд ли* выберут какой-либо из этих трех вариантов.

Подумайте, что вы собираетесь использовать повторно

Давайте еще раз взглянем на код функции `log_request`.

Очевидно, что некоторые ее части можно использовать повторно в другом коде, взаимодействующем с базой данных. Ниже мы подписали код функции, чтобы показать, какие ее фрагменты (на наш взгляд) можно использовать повторно, а какие — предназначены для решения ее основной задачи.



Простой анализ показывает, что функция `log_request` включает три группы инструкций.

- Инструкции, которые можно использовать повторно (такие как создание `conn` и `cursor`, а также вызовы `commit` и `close`).
- Инструкции, более характерные для конкретной задачи, но все же пригодные для повторного использования (например, словарь `dbconfig`).
- Инструкции, которые не удастся использовать повторно (такие как текст запроса в `_SQL` и вызов `cursor.execute`). Для других операций с MySQL, скорее всего, потребуются другие SQL-запросы с другими аргументами (или вообще без аргументов).

А что с тем импортом?

Нет, не забыли.

Выражение `import mysql.connector` не было забыто, когда мы думали о повторном использовании фрагментов кода из `log_request`.

Мы преднамеренно опустили упоминание об этой инструкции, потому что хотели обратить на нее особое внимание. Проблема вовсе не в том, что мы не хотим повторно использовать эту инструкцию, а в том, что она не может появляться в теле функции!

Будьте внимательны, размещая инструкции импорта

Несколькими страницами ранее мы упомянули, что, взглянув на код функции `log_request`, опытные программисты на Python могут неодобрительно хмыкнуть. Это объясняется присутствием строки `import mysql.connector` в теле функции. Хотя в нашей последней «Пробной поездке» все работало нормально. Так в чем же проблема?

Проблема возникает, когда интерпретатор встречает инструкцию `import` в вашем коде: он читает импортируемый модуль из файла целиком и выполняет его. Это нормально, когда инструкция `import` находится *за пределами функции*, потому что импортируемый модуль достаточно прочитать и выполнить *один раз*.

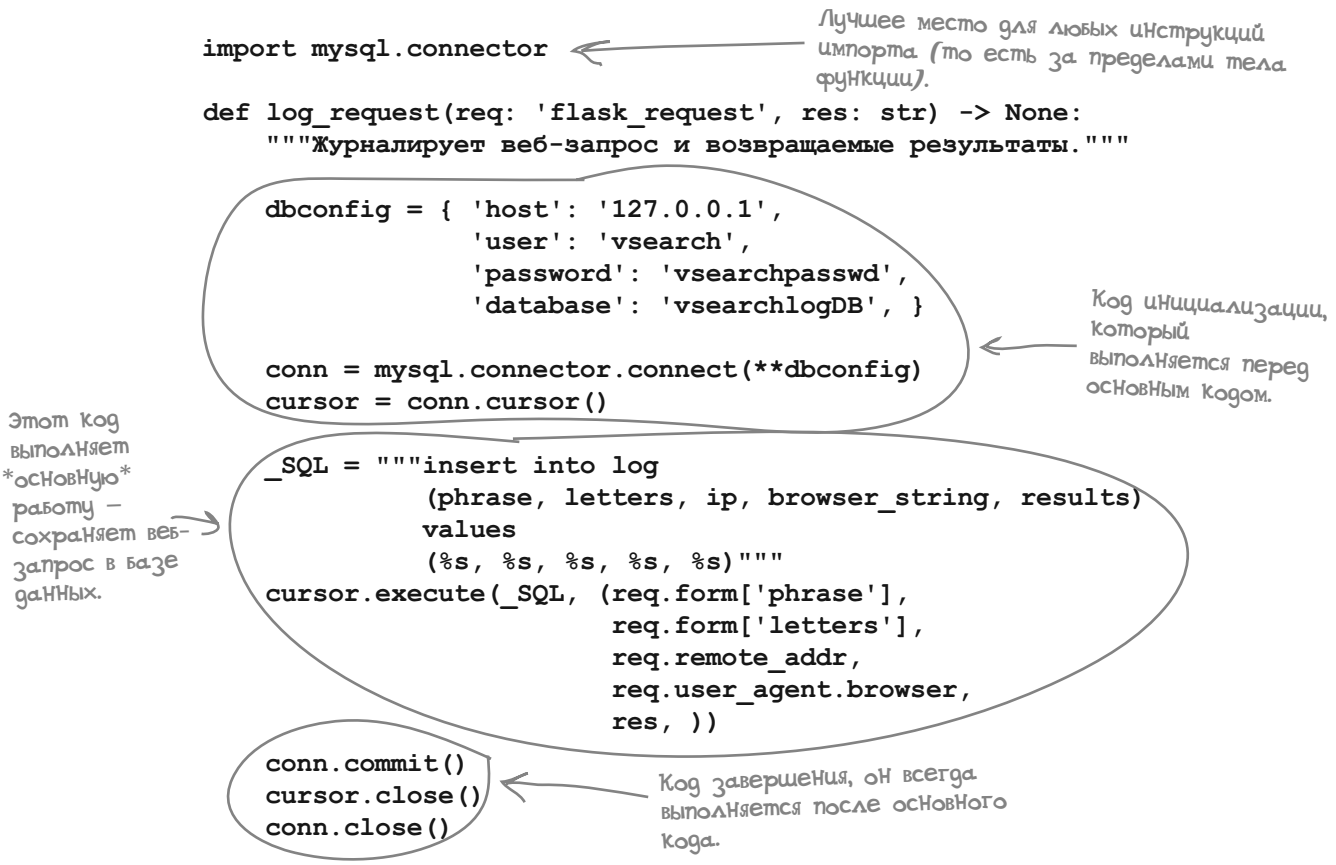
Когда инструкция `import` находится *внутри* функции, она выполняться **при каждом ее вызове**. Это чересчур расточительно (даже притом что интерпретатор — как мы видели — не запрещает использовать `import` внутри функций). Поэтому мы советуем: будьте внимательны при выборе места для инструкций `import` и не помещайте их в теле функции.

Подумайте о том, что вы собираетесь сделать

Код функции `log_request` можно рассматривать не только с точки зрения повторного использования, но и с учетом того, *когда* он выполняется.

Основой функции являются инструкция присваивания текста запроса переменной `_SQL` и вызов `cursor.execute`. Эти две инструкции определяют, *что* функция должна **делать**. Честно говоря, это наиболее важная часть. Инструкции инициализации функции определяют параметры соединения (в `dbconfig`), затем создаются соединение и курсор. Код **инициализации** всегда должен выполняться *до* основных инструкций. Последние три инструкции в функции (один вызов `commit` и два `close`) выполняются *после* основных инструкций. Этот **завершающий** код выполняет обязательные заключительные операции.

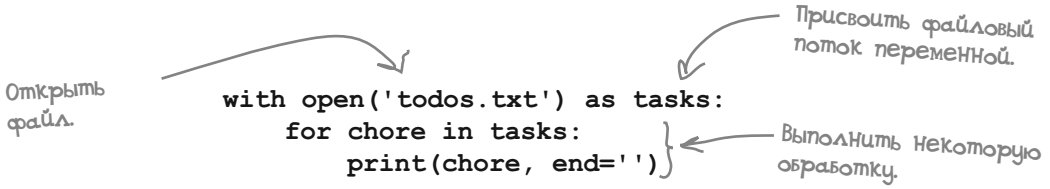
Постоянно вспоминайте шаблон — *инициализация, работа, завершение*. Теперь взгляните на функцию. Обратите внимание: сейчас инструкция `import` находится вне тела функции `log_request` (чтобы никто не хмыкал).



Хорошо бы иметь способ повторного использования шаблона: инициализация, работа, завершение!

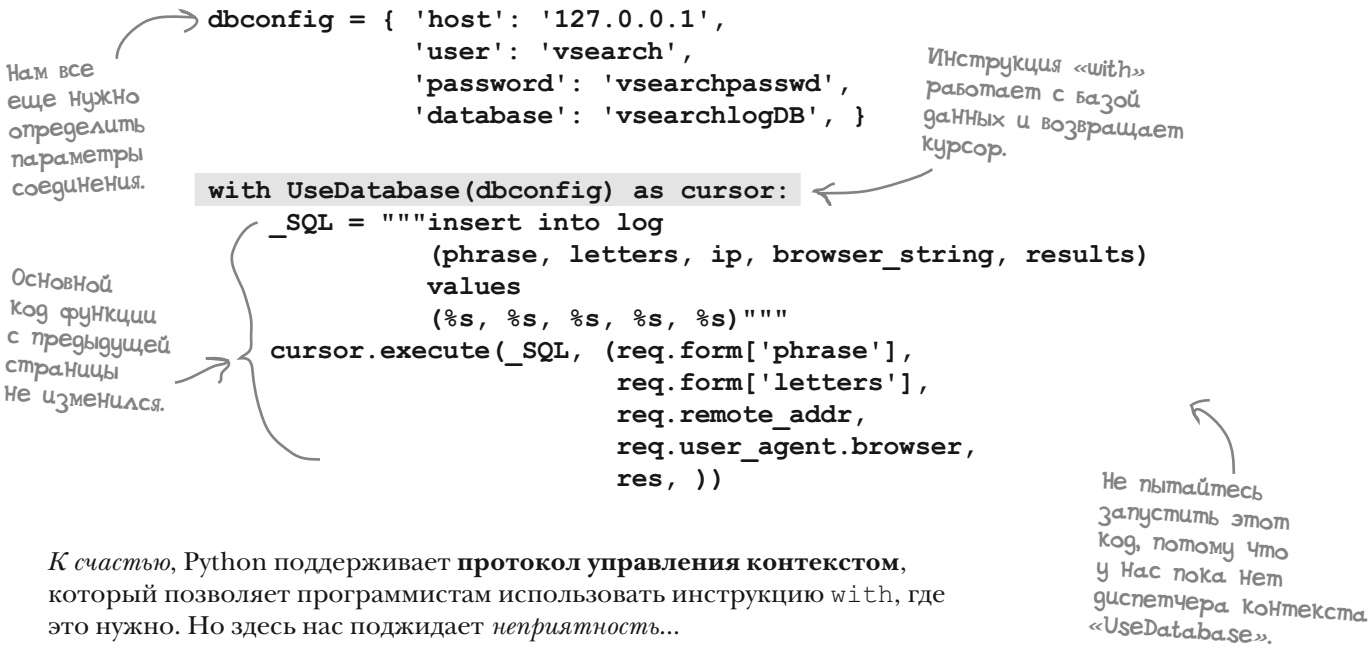
Вы видели этот шаблон раньше

Только что мы описали шаблон: код инициализации производит подготовительные операции, затем выполняется основной код и завершающий код осуществляет уборку. Возможно, это не очевидно, но в предыдущей главе подобный шаблон уже встречался.



Вспомните, как инструкция `with` *управляет контекстом*, в котором выполняется ее блок. При работе с файлами (как в примере выше) инструкция `with` открывает указанный файл и возвращает переменную, представляющую файловый поток, — `tasks` в этом примере. Это код **инициализации**. Блок инструкции `with` выполняет **основную работу** (цикл `for`, делающий «что-то важное»). Наконец, когда `with` используется для открытия файла, она гарантирует его закрытие по завершении своего блока. Это — код **завершения**.

Было бы хорошо иметь возможность поместить код для работы с базой данных в инструкцию `with`. А в идеале хотелось бы, чтобы она сама позаботилась об операциях инициализации и завершения.



К счастью, Python поддерживает **протокол управления контекстом**, который позволяет программистам использовать инструкцию `with`, где это нужно. Но здесь нас поджидает *неприятность*...

Неприятность не такая уж неприятная

В конце предыдущей страницы мы начали с *хорошей новости*: Python поддерживает протокол управления контекстом, который позволяет программистам использовать инструкцию `with`, где это нужно. Узнав, как он действует, вы сможете создать диспетчер контекста `UseDatabase`, который можно использовать совместно с инструкцией `with` для работы с базой данных.

Идея заключается в том, чтобы шаблонный код инициализации и завершения, который мы написали выше для сохранения информации в базе данных, заменить одной инструкцией `with`, как показано ниже.

```
    ...
with UseDatabase(dbconfig) as cursor:
    ...
```

Инструкция «with» похожа на ту, что мы использовали с функцией «open», но работает с базой данных.

Неприятность в том, что создание диспетчера контекста — сложный процесс, потому что нужно знать, как создать класс в Python, чтобы использовать протокол.

Подумайте, сколько полезного кода вы написали при изучении этой книги, не создав ни одного класса. Это очень хорошо, особенно если вспомнить, что в некоторых языках программирования вообще ничего нельзя сделать, не создав класс (мы смотрим на *тебя*, Java).

Однако пришло время стиснуть зубы (хотя, если честно, в создании классов в Python нет ничего ужасного).

Умение создавать классы в целом полезно, поэтому мы отложим разговоры о том, как добавить в веб-приложение код для работы с БД, и посвятим следующую (короткую) главу классам. Мы покажем ровно столько, сколько требуется для создания диспетчера контекста `UseDatabase`. После этого, в главе 9, мы вернемся к коду для работы с базой данных (и веб-приложению) и применим новые навыки в написании классов, чтобы создать диспетчер контекста `UseDatabase`.

Код из главы 7

Код для работы с базой данных, который сейчас работает внутри веб-приложения (то есть функция «log_request»).

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Журналирует веб-запрос и возвращаемые результаты."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Код, который мы хотели бы написать, чтобы сделать то же самое (заменив тело функции «log_request»). Только не пытайтесь запустить его сейчас, он не будет работать без диспетчера контекста «UseDatabase».

8 немного о Классах

Абстракция поведения и состояния

Классы позволяют связать поведение кода и состояние вместе.

В этой главе мы отложим веб-приложение в сторону и будем учиться создавать **классы**. Это умение понадобится вам при создании диспетчера контекста. Классы — настолько полезная штука, что вам в любом случае стоит ознакомиться с ними поближе, поэтому мы посвятили им отдельную главу. Мы не будем рассматривать классы во всех подробностях, а коснемся лишь тех аспектов, которые пригодятся для создания диспетчера контекста, которого ожидает наше веб-приложение.
А теперь вперед, посмотрим, что к чему.

Подключаемся к инструкции «with»

Как было сказано в конце главы 7, вы легко сможете подключить инициализирующий и завершающий код к инструкции `with`... если знаете, как создать **класс**.

Несмотря на то что половина книги осталась позади, вы успешно преодолели ее, не объявив ни одного класса. Вы написали полезный код, пригодный для многократного использования, ограничившись только механизмом функций. Но есть другие способы организации кода, и объектно-ориентированный очень популярен.

Программируя на Python, вы не обязаны пользоваться исключительно объектно-ориентированной парадигмой — этот язык обладает достаточной гибкостью и позволяет писать код в любом стиле. Но когда речь идет о подключении к инструкции `with`, делать это **рекомендуется** с помощью класса, хотя стандартная библиотека поддерживает реализацию похожей функциональности *без* классов (подход, используемый в стандартной библиотеке, применяется реже, поэтому мы не будем его использовать).

Протокол управления контекстом позволяет написать класс, который подключается к инструкции with.

Итак, для подключения к инструкции `with` необходимо создать класс. Когда вы узнаете, как писать классы, вы сможете создать свой класс и реализовать в нем поддержку **протокола управления контекстом**. Этот протокол является механизмом (встроенным в Python) подключения к инструкции `with`.

Посмотрим, как создать и использовать классы в Python, а в следующей главе обсудим протокол управления контекстом.

Это не Глупые вопросы

В: К какому типу языков программирования относится Python: объектно-ориентированному, функциональному или процедурному?

О: Это замечательный вопрос, который рано или поздно задают многие программисты, переходящие на Python. Дело в том, что Python поддерживает все эти популярные парадигмы программирования и поощряет программистов смешивать и объединять их при необходимости. Возможно, вам трудно это осознать, особенно если весь код, который вы писали раньше, был в классах, и вы создавали из них объекты (как в других языках, например в Java).

Советуем не волноваться по этому поводу: пишите код в любой парадигме, какая вам больше нравится, но не сбрасывайте со счетов другие лишь потому, что они могут — по мере приближения — выглядеть чуждыми для вас.

В: То есть... это неправильно всегда начинать с создания класса?

О: Нет, это не так — если это нужно вашему приложению. Вы не обязаны помещать весь код в классы, но если вы хотите это сделать, то Python не будет вставать у вас на пути.

До сих пор мы обходились без классов, но сейчас наступил момент, когда имеет смысл использовать класс для решения конкретной проблемы: организации повторного использования кода, взаимодействующего с базой данных. Мы смешаем и объединим парадигмы программирования, чтобы решить текущую задачу, и это хорошо.

Объектно-ориентированный пример

Прежде чем идти дальше, заметим, что мы не собирались рассматривать классы во всех подробностях. Наша цель — показать ровно столько, чтобы вы смогли уверенно создать класс, реализующий протокол управления контекстом.

Следовательно, мы не будем рассматривать темы, которые, возможно, хотелось бы обсудить закаленным практикам объектно-ориентированного программирования (ООП), такие как *наследование* и *полиморфизм* (хотя Python поддерживает и то и другое). А все потому, что при создании диспетчера контекста нас в первую очередь интересует **инкапсуляция**.

Если терминология в предыдущем абзаце вызвала у вас *безотчетную панику*, не волнуйтесь: вы можете продолжать читать, даже не понимая, что все это значит.

На предыдущей странице вы узнали, что для подключения к инструкции `with` необходимо создать класс. Прежде чем узнать, как это делается, давайте посмотрим, из чего состоит класс на языке Python, написав пример, как мы обычно делаем. Когда вы поймете, как писать класс, мы вернемся к задаче подключения к инструкции `with` (в следующей главе).

Не пугайтесь заумных словечек на этой странице!

Если бы страницы в книге соревновались, на какой из них больше всего заумных словечек, то этой странице досталась бы легкая победа. Но не нужно бояться терминов, используемых здесь. Если вы знакомы с ООП, вам все будет понятно. **Если нет, все самое важное мы расскажем и покажем на примерах.** Не волнуйтесь: многое прояснится уже на следующих нескольких страницах, по мере того, как вы будете разбирать пример.

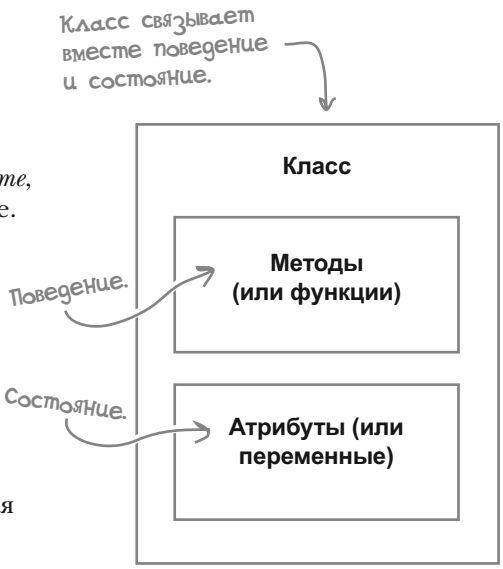
Класс связывает поведение и состояние

Классы позволяют связать **поведение** и **состояние** вместе, в один объект.

Когда вы слышите слово *поведение*, считайте, что это — *функция*, фрагмент кода, который что-то делает (*реализует поведение*, если вам так больше нравится).

Когда вы слышите слово *состояние*, считайте, что это — *переменные*, место для хранения значений внутри класса. Утверждая, что класс связывает поведение и состояние *вместе*, мы имеем в виду, что класс содержит функции и переменные.

Подведем итоги: если вы знаете, что такое функция и что такое переменные, то вы уже на полпути к пониманию того, что такое класс (и как его создать).



Классы имеют методы и атрибуты

Поведение класса в Python определяется созданием метода.

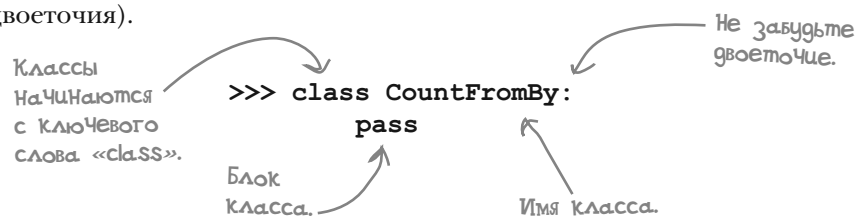
Методами в ООП называют функции, которые определяются внутри класса. Причины, по которым методы назвали методами, а не просто *функциями класса*, потерялись в тумане времени, как и причины, почему *атрибуты* назвали атрибутами, а не *переменными класса*.

Создание объектов из классов

Чтобы воспользоваться возможностями класса, нужно создать объект (как это сделать, показано в примере ниже). Эта операция называется **созданием экземпляра**. Услышав слова *создание экземпляра*, замените их мысленно словом *вызов* (то есть нужно вызвать класс, чтобы создать объект).

Как бы странно это ни звучало, можно создать класс, не имеющий состояния или поведения, который с точки зрения Python все равно считается полноценным классом. По сути это *пустой* класс. Давайте начнем пример с пустого класса и постепенно наполним его. Мы будем экспериментировать в интерактивной оболочке >>> и советуем вам повторять за нами.

Сначала создадим пустой класс с именем `CountFromBy`. Определение класса начинается с ключевого слова `class`, за которым следуют имя и блок кода, реализующий этот класс (после обязательного двоеточия).



Обратите внимание: блок этого класса состоит из единственного ключевого слова `pass`, которое в Python представляет пустую операцию (и она ничего не выполняет). Вы можете использовать `pass` в любом месте, где интерпретатор ожидает встретить настоящую инструкцию. Сейчас мы не готовы определить все детали класса `CountFromBy`, поэтому использовали `pass`, чтобы избежать любых синтаксических ошибок, которые появились бы при попытке создать класс без какого-либо кода в его блоке.

Инструкция «pass» допустима (то есть синтаксически верна), но она ничего не делает. Считайте, что это пустая операция.

Теперь у нас есть класс. Давайте создадим из него два объекта, один с именем `a` и другой — с именем `b`. Обратите внимание, что создание объекта из класса очень похоже на вызов функции.

```
>>> a = CountFromBy()
>>> b = CountFromBy()
```

Выглядит как вызовы функций, не так ли?

Создать объект, добавив круглые скобки после имени класса, а затем присвоить вновь созданный объект переменной.

Это не Глупые вопросы

В: Как, глядя в чужой код, отличить инструкцию создания объекта `CountFromBy()` от вызова функции `CountFromBy()`? Для меня она выглядит как вызов функции...

О: Отличный вопрос. По внешнему виду вы не узнаете. Однако в сообществе программистов на Python действует общепринятое соглашение: функциям дают имена, состоящие из букв нижнего регистра (с подчеркиваниями для привлечения дополнительного внимания), а классам — имена в *ВерблюжьемРегистре* (слова объединяются вместе и каждое начинается с большой буквы). Следуя этому соглашению, должно быть понятно, что `count_from_by()` — это вызов функции, а `CountFromBy()` — создание объекта. Это хорошо до тех пор, пока все следуют соглашению, и вам **настоятельно рекомендуется** поступать точно так же. Если вы проигнорируете эту рекомендацию, соглашение будет расторгнуто, и большинство программистов на Python будут держаться подальше от вас и вашего кода.

Объекты обладают общим поведением, но не состоянием

Объекты, созданные из одного класса, обладают общим поведением, реализованным в классе (совместно используют методы, определенные в классе), но поддерживают собственную копию состояния (атрибуты):

Этот различие будет иметь больше смысла, когда мы наполним пример `CountFromBy`.

Определим, что должен делать CountFromBy

Новый класс «CountFromBy» еще не создан. Мы создадим его внизу страницы.

Давайте определим, что должен делать класс `CountFromBy` (поскольку от пустого класса мало пользы).

Заставим `CountFromBy` увеличивать счетчик. По умолчанию счетчик получает начальное значение 0 и увеличивается на 1 с каждым запросом. Добавим также возможность задать другое начальное значение и/или шаг увеличения. Благодаря этому вы сможете создать, например, объект `CountFromBy`, начинающий счет со 100 и увеличивающийся с шагом 10.

Посмотрим, что класс `CountFromBy` должен делать (когда мы напишем его код). Представив, как будет использоваться класс, вы лучше поймете, как написать код `CountFromBy`. В первом примере используются умолчания класса: счет начинается с 0, и в ответ на вызов метода `increase` счетчик увеличивается на 1. Вновь созданный объект присваивается новой переменной с именем `c`.

```
>>> c = CountFromBy()
>>> c
0
>>> c.increase()
>>> c.increase()
>>> c.increase()
>>> c
3
```

Создать еще один объект и присвоить его переменной с именем «c».

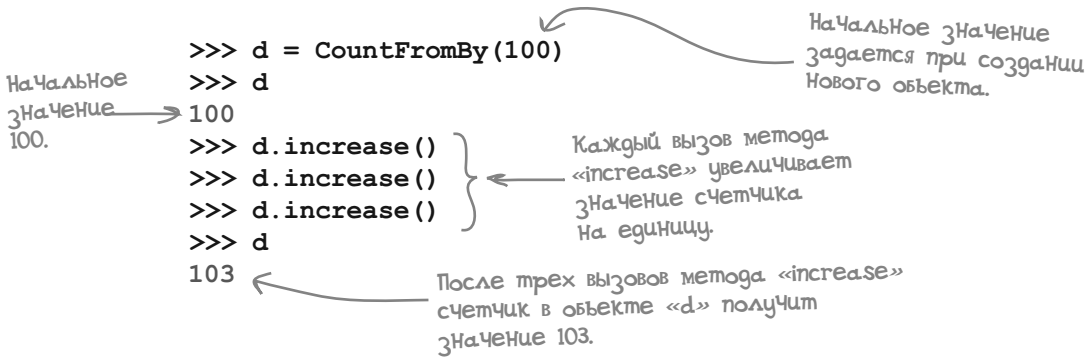
Начальное значение 0.

Каждый вызов метода «increase» увеличивает значение счетчика на единицу.

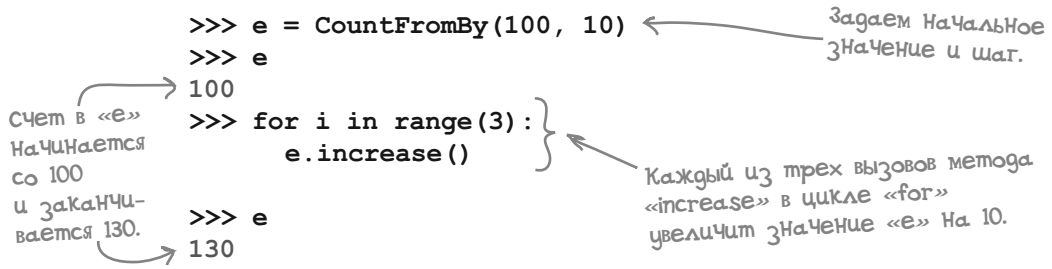
После трех вызовов метода «increase» счетчик равен трем.

Расширяем возможности CountFromBy

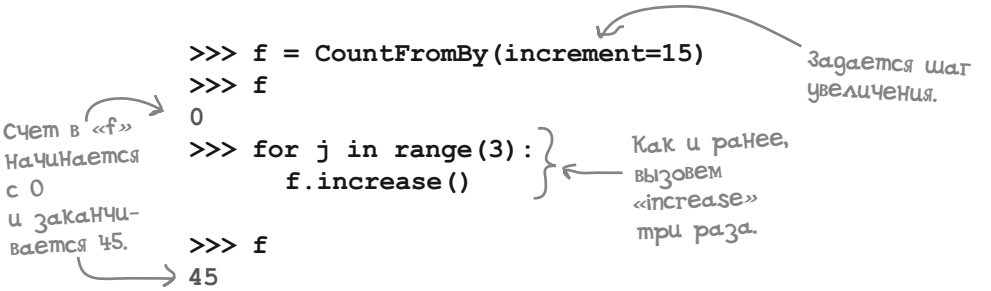
Пример использования `CountFromBy` в конце предыдущей страницы демонстрирует поведение по умолчанию; если не указано иное, счетчик, обслуживаемый объектом `CountFromBy`, начинает счет с 0 и увеличивается с шагом 1. Также можно задать альтернативное начальное значение, как показано в следующем примере, где счет начинается со 100.



Помимо начального значения, должна быть возможность задавать шаг приращения. В следующем примере счетчик получает начальное значение 100 и увеличивается с шагом 10.



В последнем примере счет начинается с 0 (по умолчанию), но с шагом 15. Вместо списка аргументов `(0, 15)` в операции создания класса в примере используется именованный аргумент, что позволяет задать шаг увеличения, оставляя начальное значение по умолчанию (0).



Повторяйте: объекты имеют общее поведение, но не состояние

В предыдущих примерах мы создали четыре новых объекта `CountFromBy`: `c`, `d`, `e` и `f`. Каждый из них имеет доступ к методу `increase`, который определяет общее поведение для всех объектов, созданных из класса `CountFromBy`. Код метода `increase` существует в единственном экземпляре и совместно используется всеми объектами. При этом каждый объект имеет свои, отдельные от других, значения атрибутов. В примерах таким значением является текущее значение счетчика, различное для каждого объекта, как показано ниже.

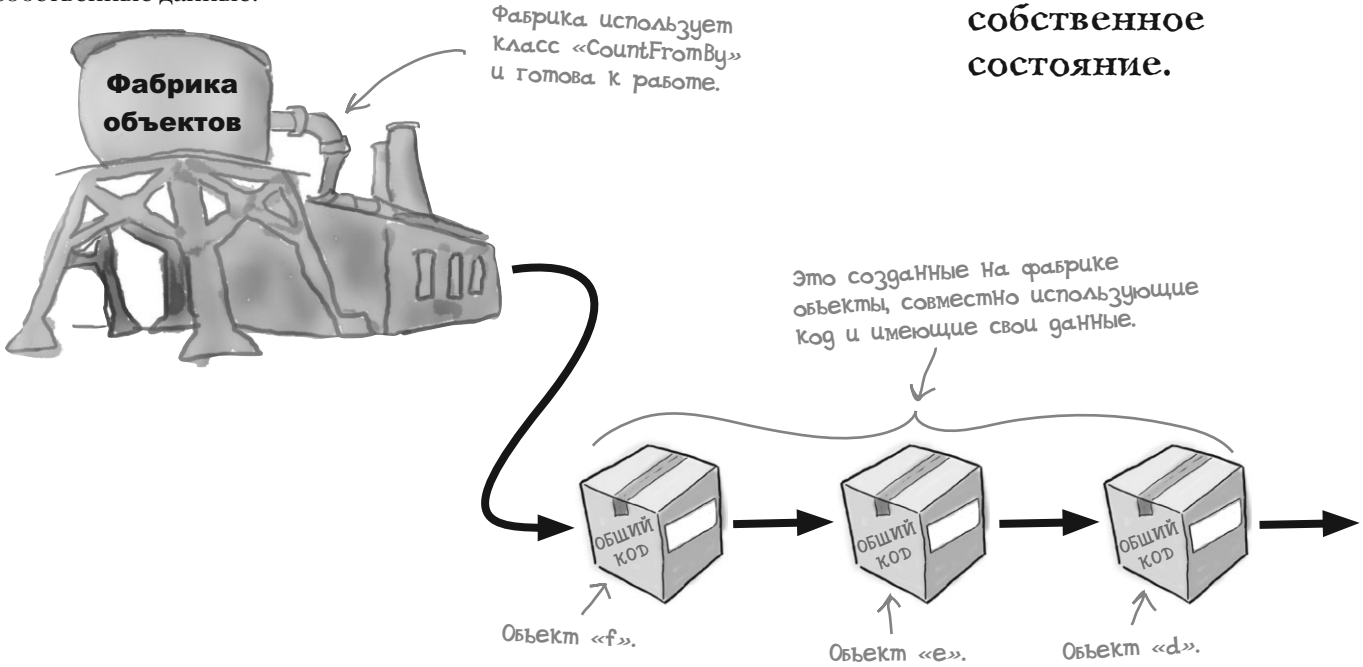
```
>>> c
3
>>> d
103
>>> e
130
>>> f
45
```

Эти четыре объекта «CountFromBy» имеют свои собственные значения атрибутов.

Поведение класса совместно используется всеми его объектами, а состояние — нет. Каждый объект имеет собственное состояние.

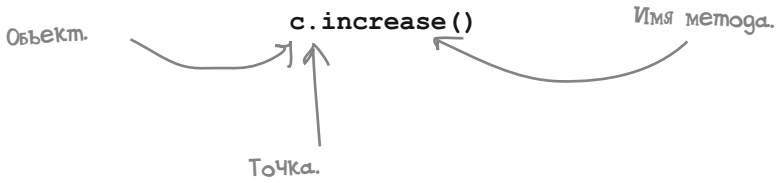
Повторим еще раз: код метода используется всеми объектами, но данные в атрибутах — нет.

Полезно думать о классе как о «форме для печенья», которая используется для создания большого количества одинаковых объектов, имеющих свои собственные данные.

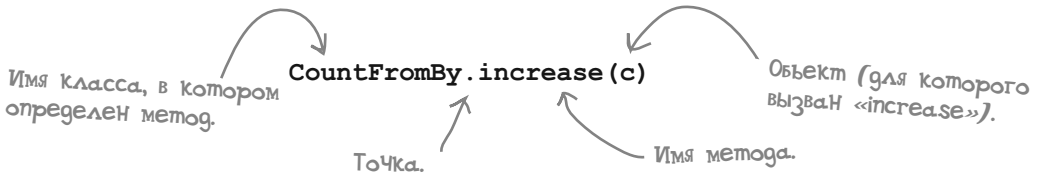


Вызов метода: подробности

Ранее мы определили метод как *функцию, объявленную внутри класса.* Мы также видели примеры вызова метода класса `CountFromBy`. Метод `increase` вызывался с использованием знакомой точечной нотации.



Будет поучительно показать код, который *на самом деле* выполняет интерпретатор (за кулисами), когда встречает такую строку. Интерпретатор развернет строку, показанную выше, в следующий код. Смотрите, что происходит с переменной `c`.



Да. Но так никто не делает.

И вы не должны так делать, потому что интерпретатор Python сделает это за вас в любом случае... Зачем писать больше кода, чтобы сделать что-то, если то же самое можно выразить более кратко?

Почему интерпретатор действует именно так, вы поймете, когда узнаете больше о том, как работают методы.

Вызов метода: что на самом деле происходит

На первый взгляд превращение интерпретатором строк `c.increase()` в `CountFromBy.increase(c)` может показаться странным, но знание происходящего поможет понять, почему каждый метод, который вы пишете, принимает *по меньшей мере* один аргумент.

Для методов нормально принимать более одного аргумента, но в первом им всегда передается сам объект (в примере на предыдущей странице — это `c`). Фактически в сообществе программистов на Python принято давать первому аргументу каждого метода специальное имя `self`.

Когда `increase` вызывается как `c.increase()`, вы можете подумать, что строка `def` с объявлением метода должна выглядеть так.

```
def increase():
```

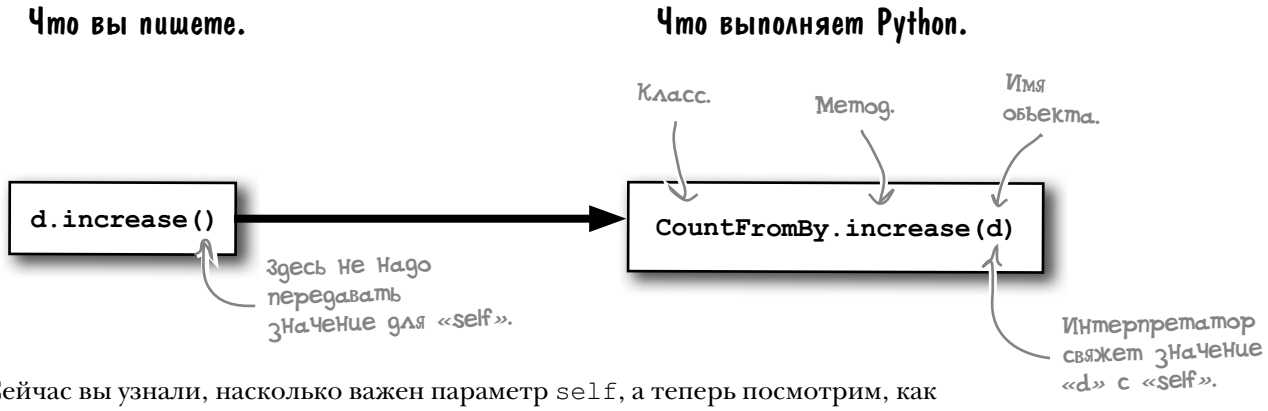
Однако определение метода без обязательного первого аргумента заставит интерпретатор сообщить об ошибке после запуска кода. Следовательно, строка `def` метода `increase` на самом деле должна быть записана так.

```
def increase(self):
```

Работая над созданием класса, воспринимайте «self» как псевдоним текущего объекта.

Считается **дурным тоном** давать первым параметрам методов имена, отличные от `self`, тем более что использование `self` требует немного времени для привыкания. (Во многих языках программирования есть схожее понятие, хотя и под другим именем `this`. Имя `self` в Python по сути означает то же самое.)

Когда программа вызывает метод объекта, Python передает ему в первом аргументе экземпляр вызывающего объекта, который *всегда* связан с параметром `self`. Этот факт объясняет особую важность параметра `self` и то, почему он всегда должен быть *первым* в любом методе объекта, который вы пишете. Вызывая метод, вы не должны передавать значение для `self` явно — интерпретатор сделает это за вас.



Сейчас вы узнали, насколько важен параметр `self`, а теперь посмотрим, как выглядит код метода `increase`.

Добавление метода в класс

Создадим новый файл, где будет храниться код нашего класса. Создайте файл `countfromby.py`, а затем добавьте в класс код, приводившийся выше в этой главе.

```
class CountFromBy:
    pass
```

Мы добавим метод `increase` в этот класс, а для этого удалим выражение `pass` и заменим его объявлением метода `increase`. Но прежде вспомним, как вызывался `increase`.

```
c.increase()
```

Опираясь на этот вызов, можно было бы предположить, что метод `increase` не принимает аргументов, так как между круглыми скобками ничего нет, верно? Однако это верно только наполовину. Как вы только что узнали, интерпретатор превращает строку кода выше в следующий вызов.

```
CountFromBy.increase(c)
```

Код метода, который мы должны написать, должен иметь в виду эту трансформацию. С учетом сказанного выше для определения метода `increase` в этом классе мы должны использовать вот какую строку `def`.

Методы по сути являются функциями, поэтому определяются с «def».

```
class CountFromBy:
    def increase(self) -> None:
```

Как и в случаях с другими функциями в этой книге, добавим аннотацию типа возвращаемого значения.

Первый параметр любого метода всегда имеет имя «self», а его значение подставляет интерпретатор.

Метод `increase` не имеет других параметров, поэтому нам не нужно добавлять в строку `def` что-то еще, кроме `self`. Однако *наличие* параметра `self` является жизненно важным условием, потому что его отсутствие приведет к синтаксическим ошибкам.

После добавления строки `def` нам осталось лишь написать некоторый код, образующий тело метода `increase`. Предположим, что этот класс поддерживает два атрибута: `val`, с текущим значением текущего объекта, и `incr`, с размером шага увеличения `val` в каждом вызове `increase`. Зная это, можно поддаться искушению и добавить в `increase` эту **неправильную** строку кода, надеясь увеличить значение атрибута

```
val += incr
```

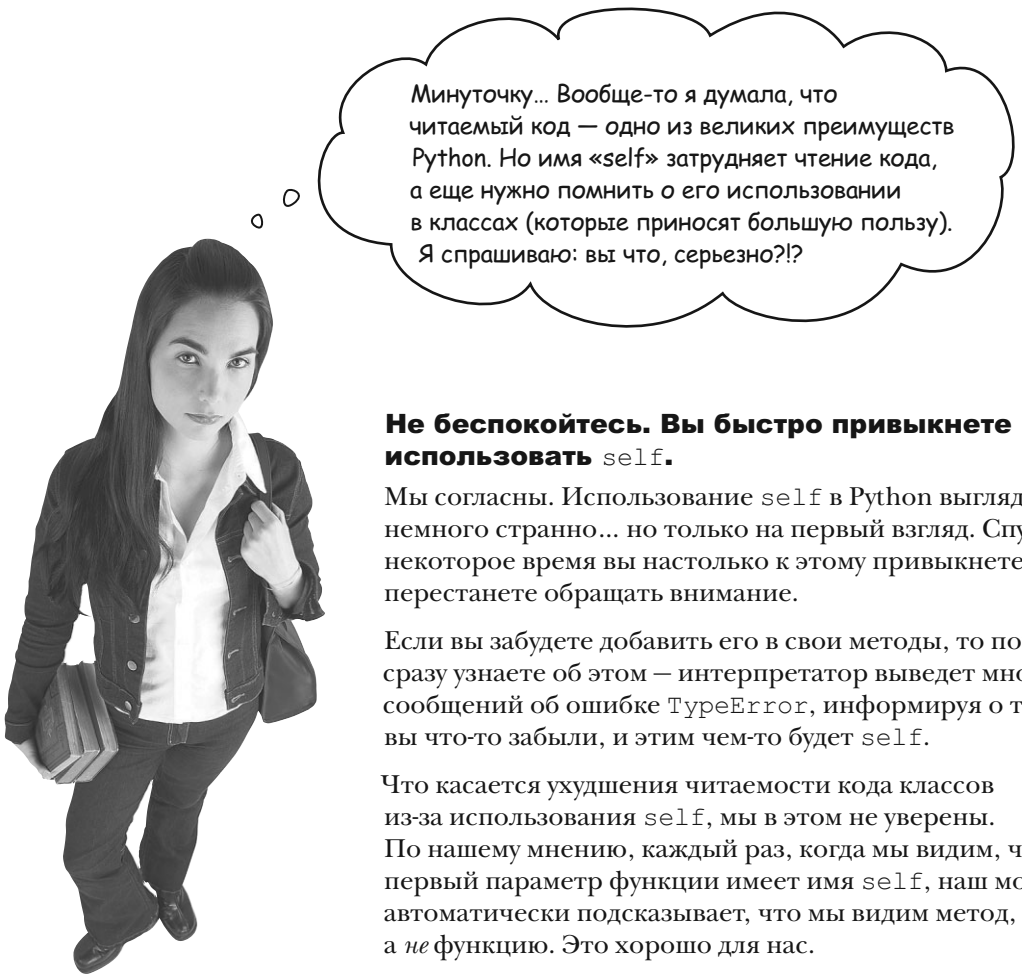
Вот как выглядит **правильная** строка кода, которая должна быть добавлена в метод `increase`.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

Берем из объекта текущее значение «val» и увеличиваем его на значение «incr».

Как вы думаете, почему эта строка кода правильная, а предыдущая нет?

О «self»: так вы серьезно?



Не беспокойтесь. Вы быстро привыкнете использовать `self`.

Мы согласны. Использование `self` в Python выглядит немного странно... но только на первый взгляд. Спустя некоторое время вы настолько к этому привыкнете, что перестанете обращать внимание.

Если вы забудете добавить его в свои методы, то почти сразу узнаете об этом — интерпретатор выведет множество сообщений об ошибке `TypeError`, информируя о том, что вы что-то забыли, и этим чем-то будет `self`.

Что касается ухудшения читаемости кода классов из-за использования `self`, мы в этом не уверены. По нашему мнению, каждый раз, когда мы видим, что первый параметр функции имеет имя `self`, наш мозг автоматически подсказывает, что мы видим метод, а *не* функцию. Это хорошо для нас.

Думайте об этом так: имя `self` сообщает, что вы читаете код метода, а не функции (в которых параметр с именем `self` *не* используется).

Важность «self»

В теле метода `increase`, приведенном ниже, все ссылки на атрибуты класса предваряются префиксом `self`. Подумайте, зачем это нужно.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

Зачем использовать «self» в теле метода?

Как вы знаете, в момент вызова метода интерпретатор связывает имя `self` с текущим объектом и поэтому ожидает, что каждый метод резервирует свой первый параметр для этой цели (чтобы присваивание выполнилось успешно).

Теперь вспомним, что мы уже знаем об объектах, созданных из класса: они совместно используют код методов класса (то есть поведение), но хранят свои *собственные копии* всех атрибутов данных (то есть состояние). Это достигается за счет связывания значений атрибутов с объектом — то есть с `self`.

Зная это, рассмотрим следующую версию метода `increase`, которая, как мы сказали пару страниц назад, **неправильная**.

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Не делайте так — этот код работает не так, как вы думаете.

Если судить по внешнему виду, последняя строка кода выглядит достаточно невинно — она просто увеличивает текущее значение `val` на текущее значение `incr`. Но посмотрите, что получится, когда метод `increase` завершится: `val` и `incr`, существующие внутри `increase`, выйдут из области видимости и, следовательно, уничтожатся, когда метод закончит работу.

Хммм... мне нужно записать «выйдут из области видимости» и «уничтожатся». Позже я посмотрю, что значат оба эти понятия... или я что-то пропустила?

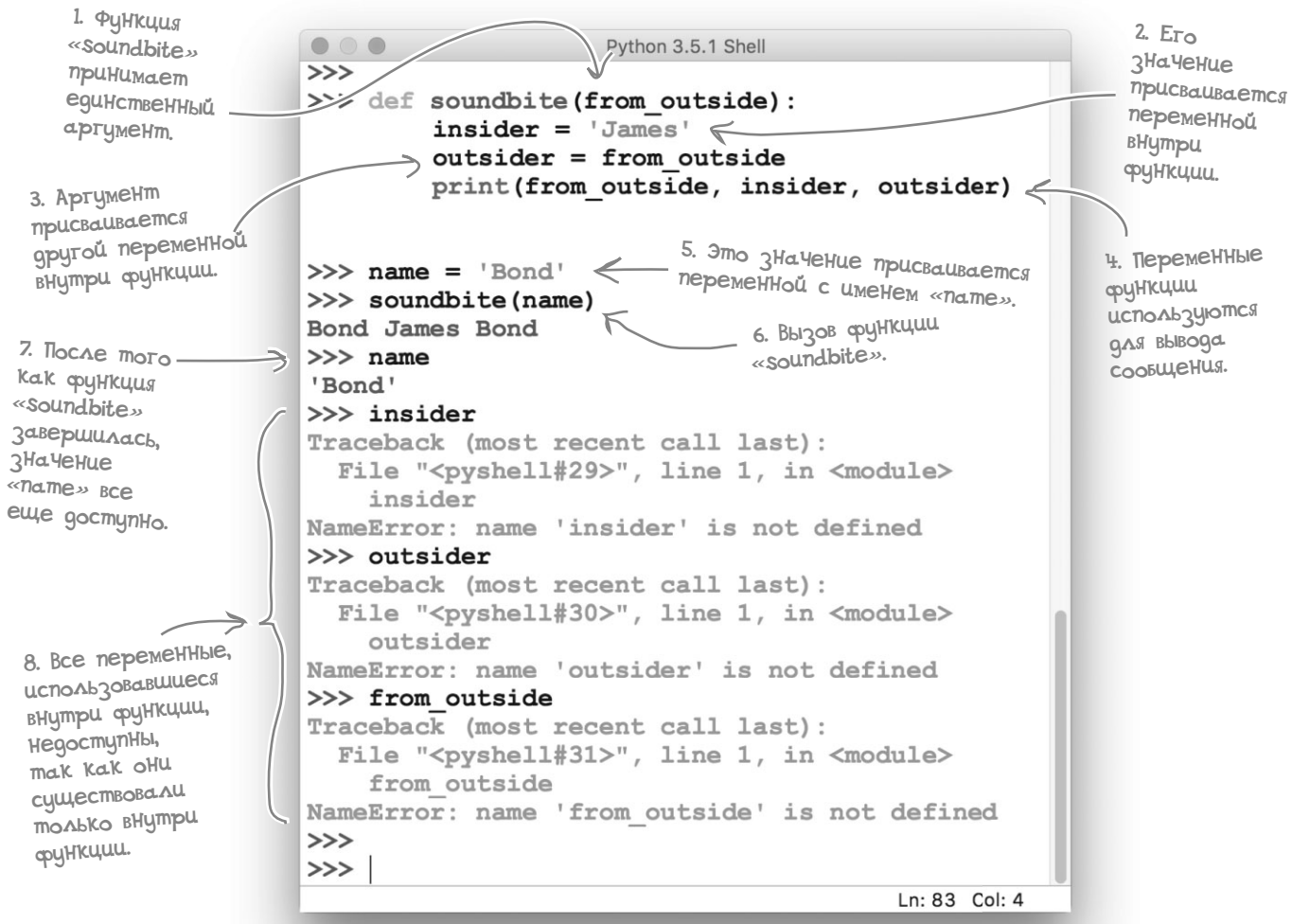
Упс. Это наша ошибка...

Кажется, мы забыли объяснить выражение «область видимости».

Чтобы вам проще было понять, что происходит, когда вы ссылаетесь на атрибуты внутри метода, давайте разберемся, что случается с переменными, используемыми в функции.

Область видимости

Посмотрим, что происходит с переменными, которые используются внутри функции, и проведем несколько экспериментов в интерактивной оболочке >>>. Опробуйте код ниже по ходу чтения. Мы пронумеровали примечания от 1 до 8, чтобы последовательно провести вас через код.



Переменные, определенные в теле функции, существуют, только пока функция выполняется. Эти переменные находятся «в области видимости», то есть видимы и могут использоваться внутри блока функции. Однако когда функция завершается, все переменные, которые были определены внутри функции, уничтожаются. Они «выходят из области видимости», и все ресурсы, которые они использовали, возвращаются интерпретатору.

Именно это и произошло с тремя переменными, использованными внутри функции `soundbite`, как было показано выше. Когда функция завершается, переменные `insider`, `outsider` и `from_outside` прекращают существование. Любая попытка обратиться к ним за пределами блока функции (то есть за пределами области видимости функции) приводит к ошибке `NameError`.

Добавляйте к именам атрибутов приставку «self»

Такое поведение функций, как на предыдущей странице, хорошо подходит для случаев, когда вы вызываете функции, они выполняют какую-то работу и возвращают значение. Вам, как правило, не важно, что происходит с переменными, использованными внутри функции, так как обычно интерес представляет только возвращаемое значение.

Теперь вы знаете, что происходит с переменными, когда функция завершается, и вам должно быть понятно, что этот (неправильный) код может привести к проблемам, если попытаться использовать переменные для сохранения и получения значений атрибутов класса. Поскольку методы — это те же функции, но с другим названием, ни `val`, ни `incr` не переживут вызова метода `increase`, если его код будет выглядеть так.

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Не делайте так, потому что переменные не выживут, когда метод завершится.

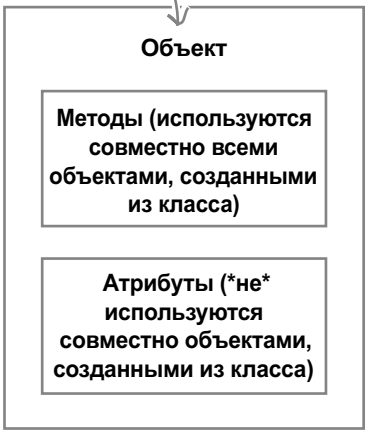
Работая с методами, надо мыслить *по-другому*. Метод использует атрибуты, принадлежащие объекту, и атрибуты объекта продолжают существовать *после* завершения метода. Атрибуты объектов **не** уничтожаются, когда метод завершается.

Чтобы значение сохранилось после завершении метода, оно должно быть связано с чем-нибудь, что не будет уничтожено, как только метод завершится. Этим *чем-нибудь* является текущий объект, для которого вызывается метод и который хранится в `self`. Вот почему в коде метода имя каждого атрибута должно предваряться приставкой `self`, как показано здесь.

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

Так намного лучше, потому что «val» и «incr» связаны с объектом благодаря использованию «self».

Псевдоним объекта — «self».



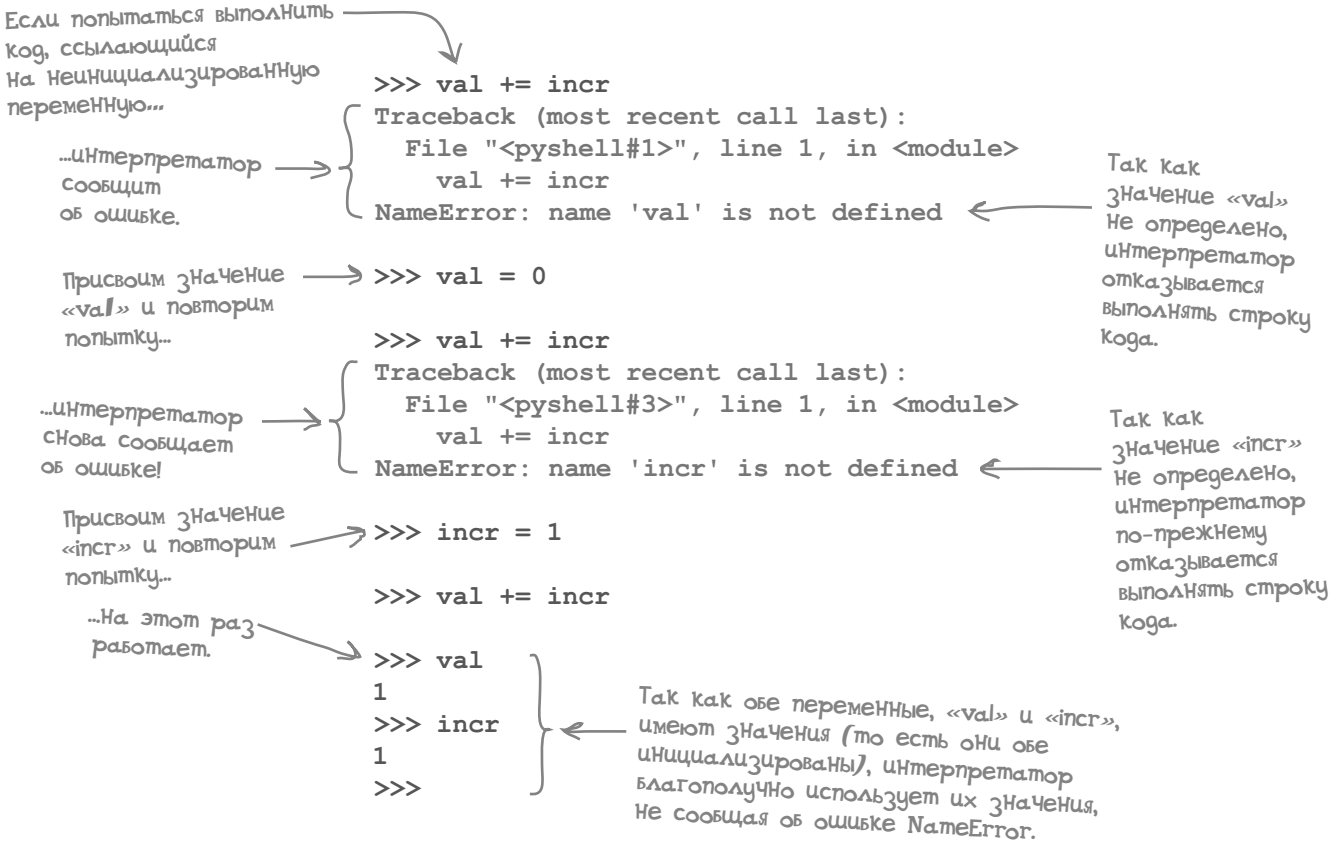
Это простое правило: если необходимо сослаться на атрибут в классе, вы *должны* предварять имя атрибута приставкой `self`. Она действует как *псевдоним* объекта, для которого вызывается метод.

В связи с этим, увидев приставку `self`, читайте ее как «этот объект». Например, `self.val` можно прочитать как «val этого объекта».

Инициализация атрибута перед использованием

Обсуждая `self`, мы оставили без внимания не менее важный вопрос: как атрибуты получают начальные значения? В текущем виде метод `increase` имеет корректный код, который использует `self`, но порождает ошибку при попытке выполнить его. Ошибка возникает из-за того, что в Python нельзя использовать переменную, пока ей не присвоено значение, независимо от того, где эта переменная используется.

Для демонстрации серьезности этого вопроса рассмотрим короткий сеанс работы в интерактивной оболочке >>>. Посмотрите, как первая инструкция терпит неудачу, если *какая-нибудь* из переменных не определена.

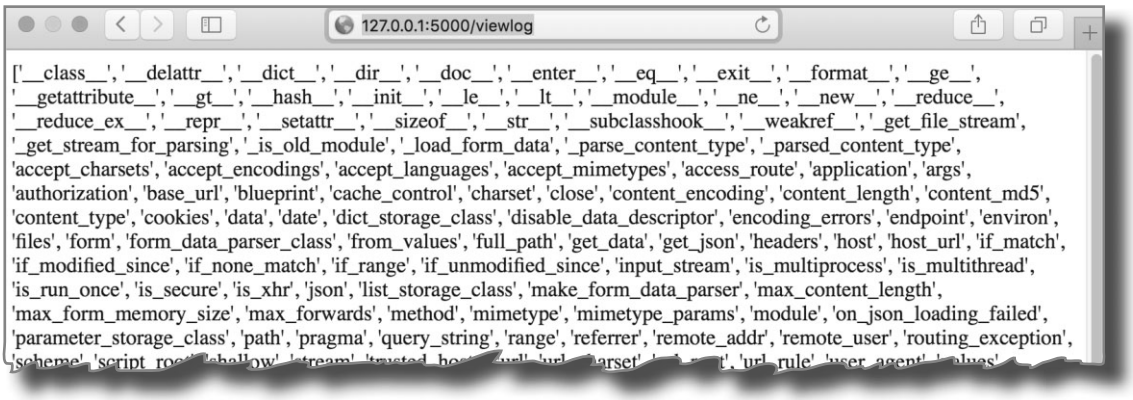


Где бы вы ни использовали переменные, вы должны инициализировать их начальными значениями. Вопрос: *как сделать это для нового объекта, созданного из класса?*

Если вы знакомы с ООП, то, возможно, прямо сейчас в вашей памяти всплыло слово «конструктор». В других языках конструктором называют специальный метод, который позволяет определить, что должно произойти в момент создания объекта. Как правило, он включает в себя создание объекта и инициализацию атрибутов. В Python интерпретатор создает объекты автоматически, поэтому для этого не нужно определять конструктор. А инициализировать атрибуты необходимыми значениями можно в специальном методе с именем `__init__`. Давайте посмотрим, что можно сделать в `init` с двойными подчеркиваниями.

Инициализация атрибутов в «init» с двойными подчеркиваниями

Давайте вспомним, о чем говорилось в предыдущей главе, когда мы использовали встроенную функцию `dir` для отображения всех деталей объекта `req` из фреймворка Flask. Помните этот вывод?



Посмотрите на все эти имена с двойными подчеркиваниями!

В тот раз мы предложили игнорировать имена с двойными подчеркиваниями. Но сейчас пора рассказать об их назначении: имена с двойными подчеркиваниями позволяют влиять на стандартное поведение класса.

Пока вы не переопределите метод с тем или иным именем, ваш класс будет использовать стандартное поведение, реализованное в классе с именем `object`. Класс `object` встроен в интерпретатор, и все другие классы (включая ваши) *автоматически* наследуют его. На языке ООП это означает, что методы класса `object` с именами, включающими двойные подчеркивания, доступны в вашем классе и их можно переопределить (чтобы заменить своей реализацией), если потребуется.

Стандартные методы с именами, включающими двойные подчеркивания и доступные для всех классов, известны как «магические методы».

Вы не должны переопределять какие-либо методы `object`, если не хотите. Но если вы захотите определить поведение объекта, созданного из вашего класса, при использовании, к примеру, с оператором сравнения (==), то можете написать свой код для метода `__eq__`. Если захотите определить поведение объекта при попытке использовать его с оператором «больше» (>), можете переопределить метод `__ge__`. А для *инициализации* атрибутов своего объекта можно использовать метод `__init__`.

Поскольку методы с именами, включающими двойные подчеркивания, реализованные в классе `object`, настолько полезны, они вызывают у программистов на Python почти мистическое благоговение. Поэтому многие из них называют эти методы *магическими методами* (кажется, что они выполняются «как по волшебству»).

Все это означает, что если определить метод `__init__`, как показано ниже в строке `def`, интерпретатор будет вызывать его при создании каждого нового объекта вашего класса. Обратите внимание на наличие первого параметра `self` в объявлении `__init__` (это правило распространяется на все методы класса).

```
def __init__(self):
```

Несмотря на странное имя, «__init__» — такой же метод, как любой другой. Запомните: в методе обязателен первый параметр с именем «self».

Инициализация атрибутов в «__init__»

Добавим __init__ в класс CountFromBy для инициализации объектов, создаваемых из нашего класса.

Для начала добавим *пустой* метод __init__, который ничего не делает, кроме pass (мы добавим поведение в ближайшее время).

```
class CountFromBy:
    def __init__(self) -> None:
        pass
    def increase(self) -> None:
        self.val += self.incr
```

Сейчас «__init__» ничего не делает. Однако наличие первого параметра с именем «self» подсказывает нам, что «__init__» — это метод.

Мы уже знаем из кода в increase, что можем получить доступ к атрибутам в нашем классе, предваряя их имена приставкой self. Этот значит, что внутри __init__ также можно использовать self.val и self.incr для ссылки на наши атрибуты. Однако мы намереваемся использовать __init__ для *инициализации* атрибутов нашего класса (val и incr). Вопрос: откуда взять эти начальные значения и как их передать внутрь __init__?

Передача любого количества аргументов в «__init__»

Так как __init__ — это метод, а методы — это замаскированные функции, вы можете передать в __init__ (или в любой другой метод, без разницы) столько аргументов, сколько пожелаете. Для этого достаточно определить параметры. Давайте дадим имя v параметру для инициализации self.val, и имя i — параметру для инициализации self.incr.

Добавим параметры v и i в строку def метода __init__, а затем используем значения в его теле для инициализации атрибутов нашего класса, как показано далее.

```
class CountFromBy:
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
    def increase(self) -> None:
        self.val += self.incr
```

Добавим параметры «v» и «i» в определение метода «__init__».

Используем значения «v» и «i» для инициализации атрибутов класса (которые представлены как «self.val» и «self.incr»).

Если теперь мы сможем сделать так, чтобы v и i принимали значения, последняя версия __init__ будет инициализировать атрибуты нашего класса. Тогда возникает еще один вопрос: как передать значения в v и i? Чтобы ответить на него, необходимо опробовать эту версию класса и посмотреть, что происходит. Давайте так и сделаем.

Пробная поездка

Используя окно редактирования IDLE, уделите минуту и обновите код в вашем файле `countfromby.py`, чтобы он выглядел, как показано ниже. Закончив, нажмите F5, чтобы начать создавать объекты в интерактивной оболочке >>> IDLE.

Нажмите F5, чтобы опробовать класс «CountFromBy» в оболочке IDLE.

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr
```

Ln: 2 Col: 0

Последняя версия класса «CountFromBy».

Нажатие F5 выполнит код, находящийся в окне редактирования, который импортирует класс `CountFromBy` в интерпретатор. Посмотрите, что происходит при попытке создать новый объект из класса `CountFromBy`.

Создаем новый объект (с именем «g») из класса... и получаем ошибку!

Python 3.5.1 Shell

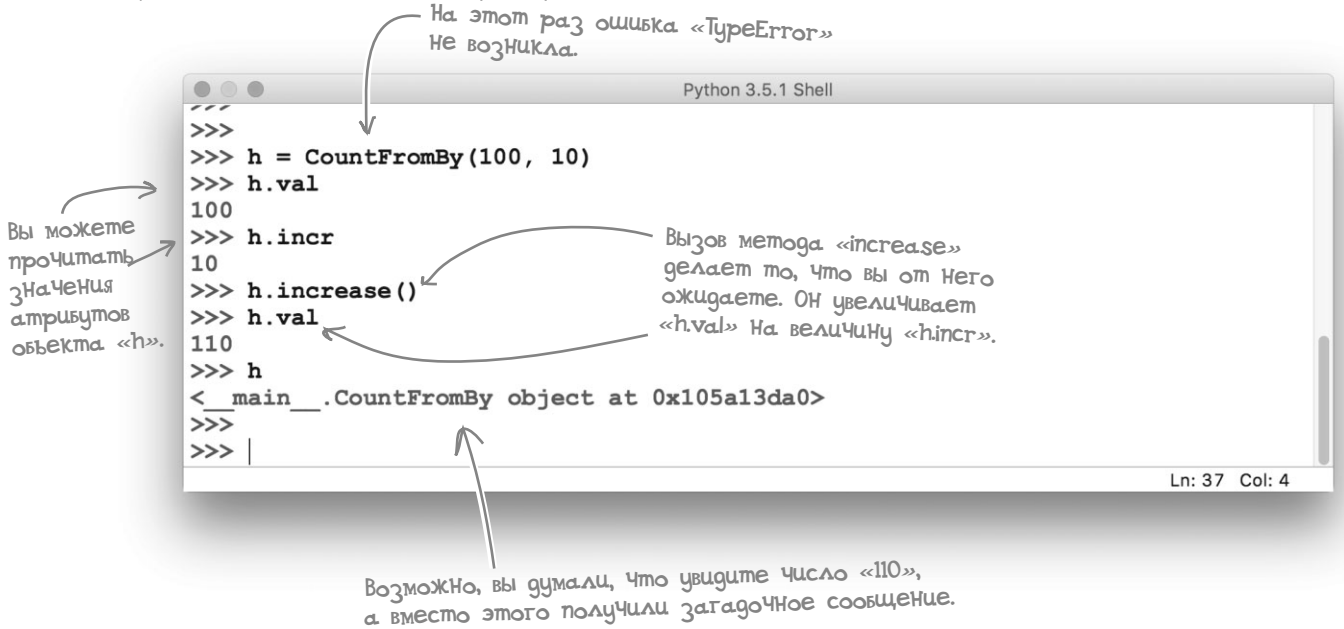
```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch07/countfromby.py =========
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Ln: 13 Col: 4

Возможно, это не то, что вы ожидали увидеть. Но посмотрите на сообщение об ошибке (которая классифицируется как `TypeError`), уделите особое внимание сообщению в строке, начинающейся с `TypeError`. Интерпретатор говорит, что метод `__init__` ожидает получить два аргумента, `v` и `i`, но получает что-то иное (в нашем случае ничего). Мы не передали аргументы классу, но сообщение об ошибке говорит нам, что любые аргументы, переданные классу (при создании объекта), передаются в метод `__init__`.

Учтем это и попробуем еще раз создать объект `CountFromBy`.

Вернемся в интерактивную оболочку >>> и создадим другой объект (с именем `h`), передав два целочисленных значения для параметров `v` и `i`.



Как видите, попытка удалась. Исключение `TypeError` пропало, значит, объект `h` был создан успешно. Вы можете обращаться к атрибутам объекта `h`, используя ссылки `h.val` и `h.incr`, а также вызывать его метод `increase`. Но при попытке получить значение `h` опять произошло что-то странное.

Что мы узнали в этой «Пробной поездке»?

Вот основные уроки, полученные в этой «Пробной поездке».

- Когда создается объект, значения всех аргументов, передаваемых классу, попадают в метод `__init__`, в примере выше это числа `100` и `10`. (Заметьте, что `v` и `i` перестанут существовать, как только `__init__` завершится, но мы не беспокоимся, так как их значения надежно сохранены в атрибутах `self.val` и `self.incr` объекта.)
- Мы можем получить значение атрибута, объединив имя объекта с именем атрибута. Обратите внимание, как мы использовали для этого `h.val` и `h.incr`. (Для читателей, пришедших в Python из «строгих» языков ООП: и мы сделали это, не создавая дополнительные методы *чтения* и *записи*.)
- Когда имя объекта использовалось само по себе (как в последней операции в примере выше), интерпретатор вывел странное сообщение. Что это было (и почему так происходит) — обсудим дальше.

Представление CountFromBy

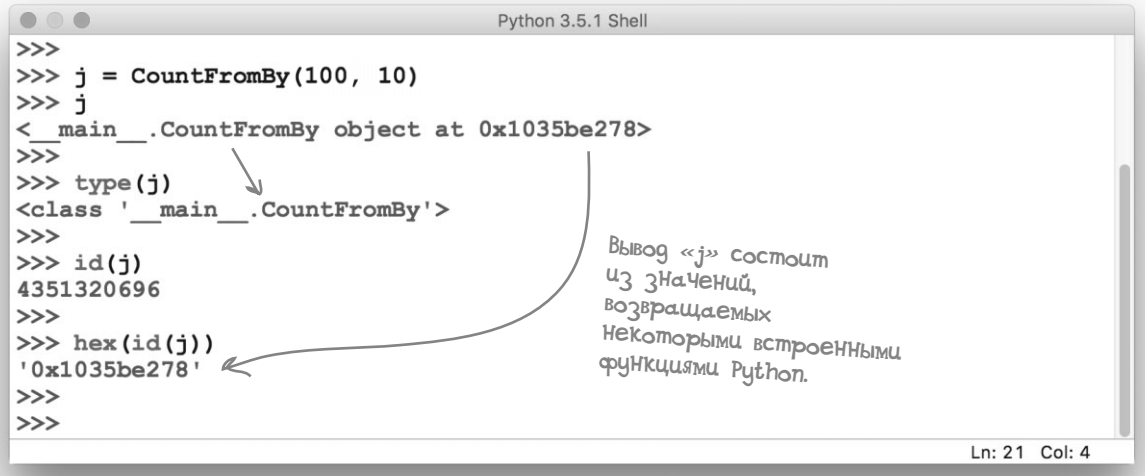
Когда мы ввели имя объекта в интерактивной оболочке, пытаясь посмотреть его текущее значение, интерпретатор вывел такую строку.

```
<__main__.CountFromBy object at 0x105a13da0>
```

Не беспокойтесь, если у вас тут другое значение. Все станет понятно прежде, чем закончится эта страница.

Мы назвали этот результат «странным», и на первый взгляд он действительно так выглядит. Чтобы понять, что этот вывод значит, вернемся в оболочку IDLE и создадим другой объект из `CountFromBy`, который, из-за нашего глубоко укоренившегося нежелания раскачивать лодку, назовем `j`.

Посмотрите, какое странное сообщение выводится для `j`. Оно включает значения, которые возвращают некоторые встроенные функции. Повторите сначала эти действия за нами, а затем прочитайте объяснение результатов вызовов этих функций.



Функция `type` возвращает информацию о классе, из которого был создан объект, сообщая (выше), что `j` — это объект `CountFromBy`.

Функция `id` возвращает информацию об адресе объекта в памяти (который используется интерпретатором в качестве уникального идентификатора для отслеживания объектов). У себя вы, скорее всего, увидите другое число, отличающееся от того, что показано выше.

Адрес в памяти, составляющий часть значения `j`, — это результат вызова функции `id`, преобразованный в шестнадцатеричное представление (это делает функция `hex`). Итак, сообщение, отражающее значение `j`, — это комбинация значений, возвращаемых функциями `type` и `id` (последнее преобразуется в шестнадцатеричный вид).

Переопределив метод «`__repr__`», можно задать, как интерпретатор должен представлять ваши объекты.

Возникает вопрос: *почему это происходит?*

Мы не сообщили интерпретатору, как хотели бы представлять ваши объекты, но интерпретатор должен *что-то* сделать, и он делает то, что показано выше. К счастью, мы можем переопределить это поведение по умолчанию, написав свой собственный магический метод `__repr__`.

Определение представления CountFromBy

Функциональность магического метода `__repr__` доступна также в виде встроенной функции `repr`. Вот что сообщает встроенная функция `help`, если попросить ее рассказать о `repr`: «Возвращает каноничное строковое представление объекта». Другими словами, `help` сообщает, что `repr` (и, соответственно, `__repr__`) должна возвращать строковое представление объекта.

Как выглядит «строковое представление объекта», зависит от самого объекта. Вы можете управлять представлением *своих* объектов, написав метод `__repr__` для своего класса. Давайте сделаем это для класса `CountFromBy`.

Сначала добавим в класс `CountFromBy` новую строку `def`, объявляющую метод `__repr__`, который не имеет параметров, кроме обязательного `self` (помните: это метод). Как обычно, добавим аннотацию. Это позволит читателям нашего кода узнать, что этот метод возвращает строку.

```
def __repr__(self) -> str:
```

При определении любого метода важно помнить, что интерпретатор всегда передает значение для первого параметра.

Так пользователи метода узнают, что функция возвращает строку. Запомните: использовать аннотации необязательно, но полезно.

Написав строку `def`, нам остается написать код, который будет возвращать строку с представлением объекта `CountFromBy`. В данном случае мы просто возьмем целочисленное значение `self.val` и преобразуем его в строку.

Благодаря функции `str`, сделать это очень просто.

```
def __repr__(self) -> str:
    return str(self.val)
```

Получить значение «self.val», превратить в строку и вернуть тому, кто вызывал метод.

После добавления в класс этой короткой функции интерпретатор будет использовать ее, когда потребуется показать объект `CountFromBy` в интерактивной оболочке `>>>`. Функция `print` также использует `__repr__` для вывода объектов.

Перед тем как внести эти изменения и применить обновленный код, вернемся ненадолго к другой проблеме, с которой мы столкнулись в последней «Пробной поездке».

Определение целесообразных умолчаний для CountFromBy

Вспомним, как выглядит текущая версия метода `__init__` класса `CountFromBy`.

```
        ...
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
        ...
```

Версия метода «__init__» ожидает передачи двух аргументов при любом вызове.

Когда мы пытались создать новый объект из этого класса, не передав значения для `v` и `i`, мы получили ошибку `TypeError`.

```
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

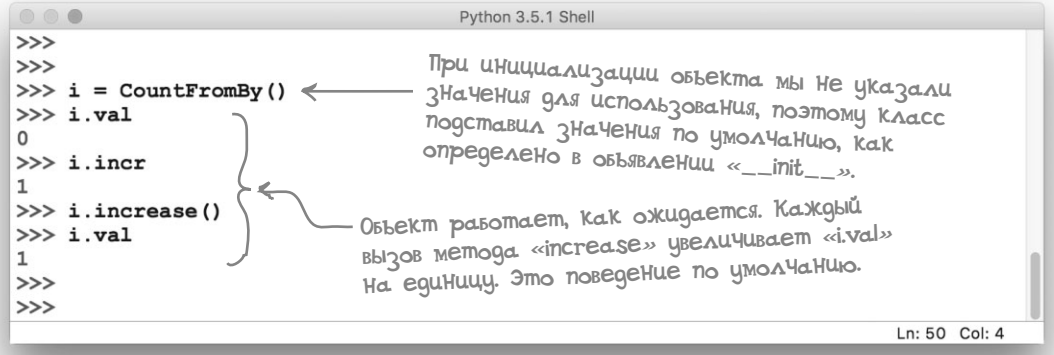
Вот она!

В начале главы мы определили, что класс `CountFromBy` должен поддерживать следующее поведение по умолчанию: счетчик начинается с `0` и увеличивается (по запросу) на `1`. Как вы знаете, чтобы определить значения по умолчанию для аргументов функции, нужно присвоить их в строке `def` — тот же прием применим к методам.

```
        ...
    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i
        ...
```

Так как методы — это функции, они поддерживают значения по умолчанию для аргументов (здесь мы присвоили значения по умолчанию параметрам с односимвольными именами: «v» — для значения, и «i» — для шага увеличения).

Если вы сделали это маленькое (но важное) изменение в коде своего класса `CountFromBy`, а затем сохранили файл (перед чем нажать F5 еще раз), то увидите, что теперь объекты могут создаваться с этим поведением по умолчанию.



Пробная поездка

Убедитесь, что код вашего класса (в `countfromby.py`) идентичен следующему. Загрузив код класса в окно редактирования IDLE, нажмите F5, чтобы взять последнюю версию `CountFromBy` на следующий круг.

Класс «CountFromBy» со своей реализацией «__repr__».

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

Ln: 13 Col: 0

Python 3.5.1 Shell

```
>>> k = CountFromBy()
>>> k
0
>>> k.increase()
>>> k
1
>>> print(k)
1
>>> l = CountFromBy(100)
>>> l
100
>>> l.increase()
>>> print(l)
101
>>> m = CountFromBy(100, 10)
>>> m
100
>>> m.increase()
>>> m
110
>>> n = CountFromBy(i=15)
>>> n
0
>>> n.increase()
>>> n
15
>>>
```

Ln: 33 Col: 4

Объект «k» использует значения по умолчанию: счет начинается с 0 и значение счетчика увеличивается с шагом 1.

Когда вы ссылаетесь на объект в командной строке >>> или вызываете «print», выполняется код «__repr__».

Объект «l» создается с альтернативным начальным значением, которое увеличивается на 1 с каждым вызовом «increase».

Объект «m» создается с альтернативными значениями для обоих атрибутов.

При создании объекта «n» используется именованный аргумент, определяющий альтернативное значение для шага увеличения (но счет начинается с 0).

Классы: что мы знаем

Теперь, создав класс `CountFromBy`, действующий в соответствии с требованиями, изложенными в начале главы, перечислим, что же мы узнали о классах в Python.

КОНТРОЛЬНЫЙ СПИСОК

- Классы в Python позволяют определять **поведение** (методы) и **состояние** (атрибуты).
- Не будет большой ошибкой считать, что методы — это **функции**, а атрибуты — **переменные**.
- Определение нового класса начинается с ключевого слова `class`.
- Создание нового объекта из класса очень похоже на вызов функции. Например, чтобы создать объект с именем `mycount` из класса `CountFromBy`, следует использовать эту строку кода: `mycount = CountFromBy()`.
- Код класса **совместно** используется всеми объектами, созданными из этого класса. Но каждый объект имеет свою **собственную копию** атрибутов.
- Поведение в класс добавляется путем создания **методов**. Метод — это функция, определенная внутри класса.
- Чтобы добавить **атрибут** в класс, создайте переменную.
- Каждому методу в первом параметре передается **псевдоним** текущего объекта. В соответствии с общепринятым соглашением, этот параметр должен иметь имя `self`.
- В теле метода ссылки на атрибуты должны предваряться приставкой `self`, что обеспечит **выживание** значения атрибута после завершения работы метода.
- Метод `__init__` — это один из множества **магических методов**, наследуемых всеми классами в Python.
- Атрибуты инициализируются в методе `__init__`. Этот метод позволяет присвоить начальные значения атрибутам в момент создания нового объекта. Он получает **копии** всех аргументов, переданных в вызов класса при создании объекта. Например, при создании следующего объекта метод `__init__` получит значения `100` и `10`:
  `mycount2 = CountFromBy(100, 10)`
- Другой магический метод — `__repr__` — позволяет управлять представлением объекта при отображении в интерактивной оболочке `>>>` или с помощью встроенной функции `print`.

Мы хотели создать диспетчер контекста.
Да, нам пришлось немного отвлечься, но мы встали на путь изучения классов, чтобы узнать все необходимое для реализации **протокола управления контекстом**. Сумев реализовать протокол, мы используем код для работы с базой данных в нашем веб-приложении с инструкцией `with`, что упростит его повторное использование. Сейчас, после знакомства с классами, вы готовы приступить к реализации протокола управления контекстом (в главе 9).

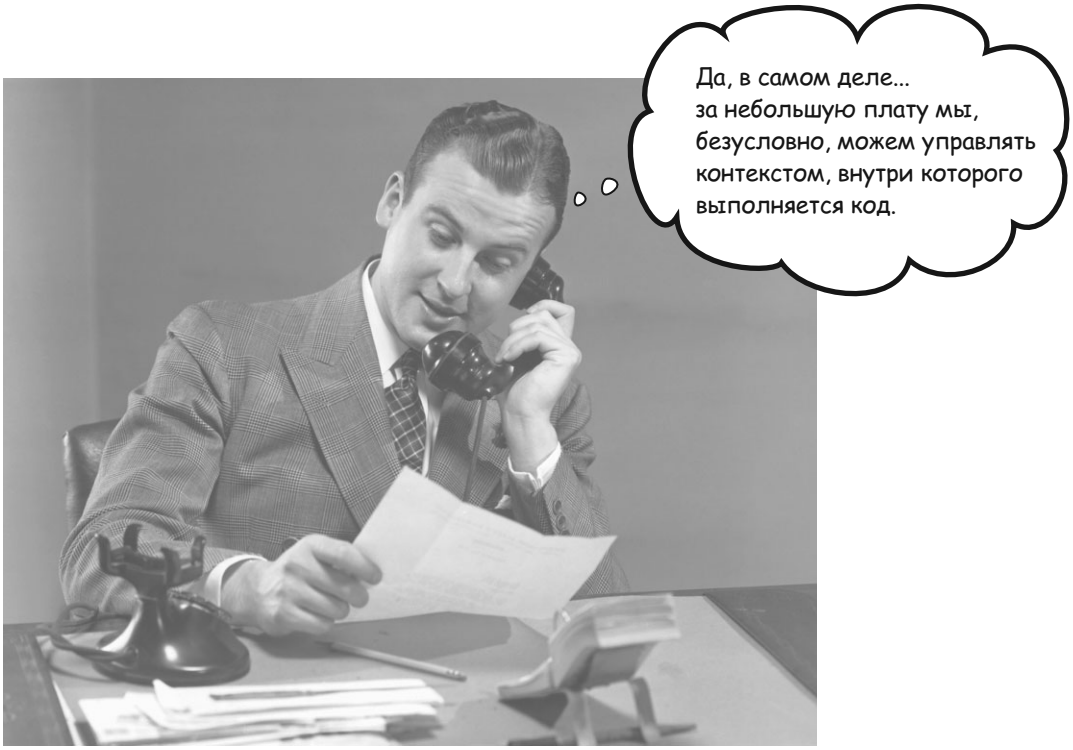
Код из главы 8

Код в файле «countfromby.py».

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

9 протокол управления контекстом

Подключение к инструкции with



Пора применить все изученное на практике.

В главе 7 мы обсудили использование **реляционных баз данных** в Python, а в главе 8 ознакомились с использованием **классов**. В главе 9 мы объединим обе методики и создадим **диспетчер контекста**, который позволит расширить инструкцию `with` для работы с системами реляционных баз данных. В этой главе вы подключитесь к инструкции `with`, создав новый класс, соответствующий требованиям **протокола управления контекстом** в языке Python.

Выбираем лучший способ повторного использования кода для работы с базой данных

В главе 7 мы создали действующую функцию `log_request` для работы с базой данных, но нам пришлось отложить выбор лучшего способа повторного использования кода. Вспомним варианты, предложенные в конце главы 7.

Тогда мы сказали, что каждое из этих предложений по-своему правильное, но отметили, что программисты на Python вряд ли обрадуются *любому из них*. Мы решили, что лучшая стратегия — реализовать протокол управления контекстом и использовать инструкцию `with`, но для этого нужно было ознакомиться с классами, что мы и сделали в главе 8. Сейчас, когда вы знаете как создавать классы, пора вернуться к поставленной задаче и создать диспетчер контекста для работы с базами данных внутри веб-приложения.

Вспомним, что мы собирались сделать

Ниже приведен код для работы с базой данных из главы 7. Сейчас он является частью нашего веб-приложения. Вспомним, как этот код подключался к базе данных MySQL, сохранял информацию о веб-запросе в таблицу `log`, подтверждал запись *несохраненных* данных, а затем отключался от базы данных.

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Журналирует веб-запрос и возвращаемые результаты."""

    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

Словарь с параметрами подключения к базе данных.

Фрагмент использует учетные данные для подключения к базе данных и создает курсор.

Код делает основную работу: добавляет информацию о запросе в таблицу «log».

Наконец, этот код закрывает соединение с базой данных.

Как лучше создать диспетчер контекста?

Прежде чем превратить код в нечто, что можно использовать как часть инструкции `with`, обсудим, как реализуется поддержка протокола управления контекстом. Стандартная библиотека позволяет создавать простые диспетчеры контекста (с использованием модуля `contextlib`), но когда предполагается использовать `with` для управления некоторым внешним объектом, таким как соединение с базой данных (наш случай), правильным подходом считается создание класса, реализующего протокол.

Принимая это во внимание, посмотрим, что же означает фраза «реализация протокола управления контекстом».

Управление контекстом с помощью методов

Протокол управления контекстом, хоть и кажется пугающим, на самом деле очень прост. Он требует, чтобы класс, реализующий протокол, определял по меньшей мере два магических метода: `__enter__` и `__exit__`. Это и есть протокол. Класс, соответствующий этим требованиям, можно подключить к инструкции `with`.

Метод «__enter__» выполняет настройку

Когда инструкции `with` передается объект, интерпретатор вызывает его метод `__enter__` *перед* тем, как начать выполнять тело инструкции `with`. Это дает возможность выполнить любые необходимые настройки внутри `__enter__`.

Затем протокол заявляет, что `__enter__` может (но не обязан) вернуть значение для инструкции `with` (вскоре вы узнаете, почему это важно).

Протокол определяет процедуру (или набор правил), которой следует придерживаться.

Метод «__exit__» выполняет завершающий код

Закончив выполнение тела инструкции `with`, интерпретатор *обязательно* вызывает метод объекта `__exit__`. Это происходит *после* завершения блока `with`, что позволяет выполнять любой необходимый завершающий код.

Так как код в теле инструкции `with` может завершиться аварийно (и возбудить исключение), `__exit__` должен быть готов обработать эту ситуацию. Мы вернемся к этому вопросу, когда создадим код для нашего метода `__exit__`.

Если определить класс с методами `__enter__` и `__exit__`, он автоматически будет восприниматься интерпретатором как диспетчер контекста и сможет подключаться к инструкции `with` (и использоваться с ней). Другими словами, такой класс *соответствует требованиям* протокола управления контекстом и *реализует* диспетчер контекста.

Если класс определяет методы «__enter__» и «__exit__» — это диспетчер контекста.

(Как известно) «__init__» выполняет инициализацию

Кроме `__enter__` и `__exit__`, в класс можно добавить другие методы, включая метод `__init__`. Как вы узнали в предыдущей главе, метод `__init__` позволяет выполнять дополнительную инициализацию объекта. Он вызывается *перед методом* `__enter__` (то есть *до того как будет выполнен код настройки диспетчера контекста*).

Наличие метода `__init__` в диспетчере контекста — необязательное требование (достаточно определить только методы `__enter__` и `__exit__`), но иногда полезно его добавить, поскольку так можно отделить все операции по инициализации действий по настройке. Когда мы приступим к созданию диспетчера контекста (далее в главе) для управления соединением с базой данных, мы определим метод `__init__`, инициализирующий учетные данные для соединения. Делать так абсолютно не обязательно, но мы полагаем, что это поможет сохранить код диспетчера контекста ясным и аккуратным, а также простым для чтения и понимания.

Вы уже видели, как действует диспетчер контекста

Мы впервые встретились с инструкцией `with` в главе 6, когда использовали ее, чтобы *автоматически* закрыть предварительно открытый файл после завершения блока инструкции `with`. Вспомним, как этот код открывал файл `todos.txt`, читал и отображал его содержимое строку за строкой, после чего автоматически закрывал файл (благодаря тому факту, что `open` — это диспетчер контекста).

Наша первая инструкция «with» (из главы 6).

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

Взглянем еще раз на эту инструкцию `with`, выделив места, где вызываются `__enter__`, `__exit__` и `__init__`. Мы пронумеровали комментарии, чтобы помочь вам понять порядок выполнения методов. Обратите внимание: мы не видим здесь кода инициализации, настройки или завершения; мы просто знаем (и верим), что эти методы выполняются «за кулисами», когда это необходимо.

1. Встретив эту инструкцию «with», интерпретатор сначала выполнит «__init__», связанный с вызовом «open».

2. Как только «__init__» выполнится, интерпретатор вызовет «__enter__», чтобы присвоить результат вызова «open» переменной «tasks».

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

3. Когда тело инструкции «with» закончит выполняться, интерпретатор вызовет метод «__exit__» диспетчера контекста для корректного завершения работы. В примере интерпретатор гарантирует, что открытый файл будет корректно закрыт, перед тем как выполнение будет продолжено.

Что от вас требуется

Прежде чем перейти к созданию диспетчера контекста (с помощью нового класса), коротко перечислим требования протокола управления контекстом, которые мы должны выполнить для подключения к инструкции `with`. Мы должны создать класс, в котором определены:

1. метод `__init__` для выполнения инициализации (если необходимо);
2. метод `__enter__` для выполнения настройки;
3. метод `__exit__` для выполнения завершающих операций (то есть для уборки).

Вооружившись этим знанием, создадим класс диспетчера контекста, написав методы по очереди и заимствуя по мере необходимости существующий код для работы с базой данных.

Создание нового класса диспетчера контекста

Для начала дадим новому классу имя. Затем поместим код нового класса в отдельный файл, так мы сможем упростить его повторное использование (помните: когда код на Python помещается в отдельный файл, он становится модулем, который при необходимости можно импортировать в другие программы).

Назовем новый файл `DBcm.py` (сокращенно от *database context manager — диспетчер контекста базы данных*), а новому классу дадим имя `UseDatabase`. Создайте файл `DBcm.py` в той же папке, где сейчас находится код веб-приложения, потому что веб-приложение будет импортировать класс `UseDatabase` (как только мы напишем его).

Запомните: давая имена классам в Python, используйте ВерблюжийРегистр.

Используя свой любимый редактор (или IDLE), создайте новый пустой файл и сохраните его с именем `DBcm.py`. Мы знаем, что для соответствия требованиям протокола управления контекстом наш класс должен:

1. иметь метод `__init__`, выполняющий инициализацию;
2. иметь метод `__enter__`, выполняющий настройки;
3. иметь метод `__exit__`, выполняющий завершающие операции.

Теперь добавим в класс «пустые» определения трех обязательных методов. Пустой метод содержит единственную инструкцию `pass`. Вот код.

Вот как наш файл «DBcm.py» выглядит в IDLE. Сейчас он содержит одну инструкцию «import», а также определение класса «UseDatabase» с тремя «пустыми» методами.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

```
import mysql.connector


class UseDatabase:

    def __init__(self):
        pass

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

Обратите внимание, что в верхней части файла `DBCm.py` присутствует инструкция `import`, которая подключает драйвер *MySQL Connector* (он потребуется в нашем новом классе).

Теперь нам осталось перенести соответствующие фрагменты кода из функции `log_request` в соответствующие методы класса `UseDatabase`. Что ж... когда мы говорили *мы*, на самом деле мы подразумевали **вы**. Пора уже вам закатать рукава и написать немного кода.

Инициализация класса параметрами соединения с базой данных

Давайте вспомним, как мы намерены использовать диспетчер контекста `UseDatabase`. Ниже приведен код из главы 7, переписанный с использованием инструкции `with`, где задействован диспетчер контекста `UseDatabase` (его-то вы и должны написать).

Импортирование диспетчера контекста из файла «DBcm.py».

```
from DBcm import UseDatabase

dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

with UseDatabase(dbconfig) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Параметры подключения к базе данных.

Диспетчер контекста возвращает «cursor».

Диспетчер контекста «UseDatabase» ожидает получения словаря с параметрами для подключения к БД.

Код остается таким же, каким был до этого.

Заточите карандаш

Начнем с метода `__init__`, который мы используем для инициализации атрибутов класса `UseDataBase`. Опираясь на сценарий использования, показанный выше, метод `__init__` принимает один аргумент — словарь с параметрами подключения — имеющий имя `config` (вы должны добавить его в строку `def` ниже). Давайте условимся, что содержимое `config` будет сохранено в атрибуте с именем `configuration`. Добавьте необходимый код в `__init__`, чтобы сохранить словарь в атрибуте `configuration`.

```
import mysql.connector

class UseDatabase:

    def __init__(self, ..... )

        ..... .....
```

Закончите строку «def».

Здесь что-то пропущено?

Сохранить словарь с параметрами в атрибуте.

Заточите карандаш
Решение

Мы начали с метода `__init__`, инициализирующего атрибуты класса `UseDataBase`. Метод `__init__` принимает единственный аргумент — словарь с параметрами подключения — с именем `config` (который необходимо добавить в строку `def` ниже). Мы должны были сохранить содержимое `config` в атрибуте с именем `configuration`. Для этого мы добавили в `__init__` код, сохраняющий словарь в атрибуте `configuration`.

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict ) -> None :
        self.configuration = config
```

Метод «__init__» принимает единственный словарь, который мы назвали «config».

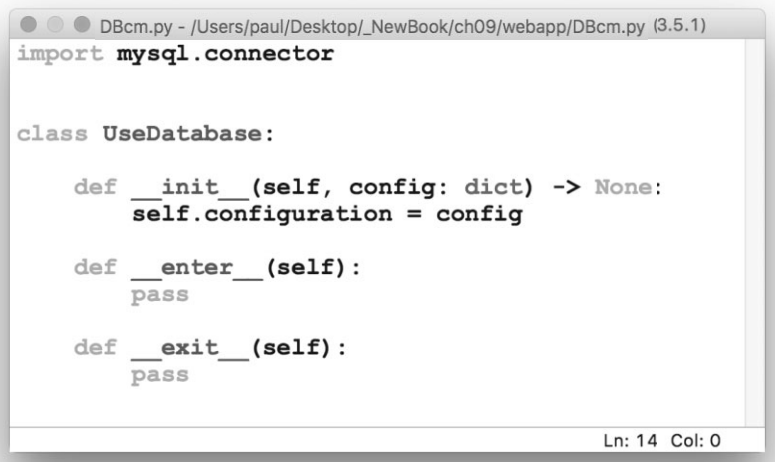
Значение аргумента «config» присваивается атрибуту «configuration». Вы еще помните, что к имени атрибута надо добавлять приставку «self»?

Необязательная аннотация «None» указывает, что метод не возвращает значение (это приятно знать), и двоеточие завершает строку «def».

Диспетчер контекста начинает приобретать форму

Закончив метод `__init__`, можно переключиться на метод `__enter__`. Перед тем как вы это сделаете, проверьте код, который вы написали. Он должен соответствовать коду, который показан ниже в окне редактирования IDLE.

Убедитесь, что ваш метод «__init__» соответствует нашему.



Выполняем настройку в «__enter__»

Метод `__enter__` обеспечивает место для кода, выполняющего настройку объекта *перед* началом выполнения блока инструкции `with`. Вспомним код из функции `log_request`, который выполняет настройку.

Вот код настройки из функции «log_request».

```
...
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

conn = mysql.connector.connect(**dbconfig)
cursor = conn.cursor()

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
       ...
```

Этот код использует словарь с параметрами, чтобы подключиться к MySQL, а затем создает курсор базы данных (который понадобится для отправки команд базе данных из нашего кода на Python). Поскольку код настройки должен выполняться при каждом подключении к базе данных, сделаем так, чтобы эту работу выполнял класс диспетчера контекста — так мы сможем упростить его повторное использование.

Заточите карандаш

Метод `__enter__` должен подключиться к базе данных, используя параметры конфигурации, хранящиеся в `self.configuration`, и затем создать курсор. Кроме обязательного аргумента `self`, метод `__enter__` не принимает ничего, но должен вернуть курсор. Завершите код метода ниже.

Добавьте сюда код настройки.

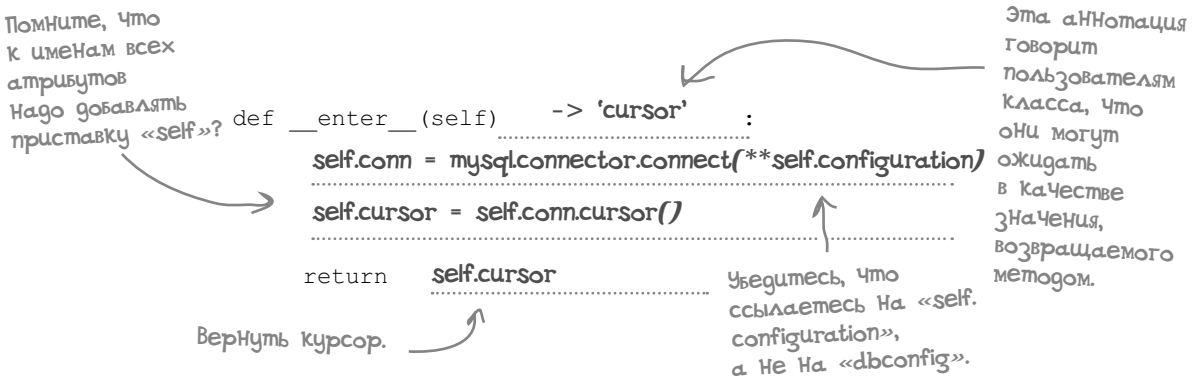
Сможете придумать соответствующую аннотацию?

```
def __enter__(self)..........:
    ..........
    ..........
    return ..........
```

Не забудьте вернуть курсор.

Заточите карандаш
Решение

Метод `__enter__` должен подключиться к базе данных, используя параметры конфигурации, хранящиеся в `self.configuration`, и затем создать курсор. Кроме обязательного аргумента `self`, метод `__enter__` не принимает ничего, но должен вернуть курсор. Вы должны были завершить код метода.



Не забывайте предварять атрибуты приставкой «self»

Возможно, вас удивило, что мы использовали переменные `conn` и `cursor` в методе `__enter__` как атрибуты (добавив к каждой приставку `self`). Мы сделали это, чтобы гарантировать их сохранность по завершении метода, так как обе они понадобятся в методе `__exit__`. Мы добавили префикс `self` к обеим переменным, `conn` и `cursor`, тем самым включив их в список атрибутов класса.

Перед тем как перейти к методу `__exit__`, проверьте свой код — он должен соответствовать нашему.

Почти закончили. Осталось написать еще один метод.

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)*

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self):
        pass
```

Ln: 16 Col: 0

Выполнение завершающего кода с использованием «__exit__»

Метод `__exit__` служит местом для кода, который должен выполниться по завершении тела инструкции `with`. Вспомним код из функции `log_request`, который выполняет завершающие действия.

```
            ...
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))

conn.commit()
cursor.close()
conn.close()
```

Завершающий код.

Завершающий код подтверждает запись данных в базу данных, а затем закрывает курсор и соединение. Завершающий код должен выполняться после *каждого* взаимодействия с базой данных, поэтому добавим его в класс диспетчера контекста, поместив эти три строки в метод `__exit__`.

Нужно сказать, что с методом `__exit__` связана одна сложность: он должен обрабатывать любые исключения, которые возникают внутри блока `with`. Когда что-то идет не так, интерпретатор *всегда* оповещает `__exit__`, передавая методу три аргумента: `exec_type`, `exc_value` и `exc_trace`. Строка `def` должна это учитывать, вот почему в коде ниже мы добавили три аргумента. Договоримся, что сейчас мы будем *игнорировать* механизм обработки исключений, но вернемся к нему в главе 10, когда будем обсуждать, что может пойти не так и как с этим справиться (поэтому продолжайте читать).

Заточите карандаш

Завершающий код выполняет уборку и наводит порядок. Для этого диспетчера контекста навести порядок означает подтвердить запись всех данных в базу, после чего закрыть курсор и соединение. Добавьте код, который, как вы полагаете, понадобится в методе, представленном ниже.

Пока не думайте об этих аргументах.

```
def __exit__(self, exc_type, exc_value, exc_trace) ..... :
```

Добавьте сюда завершающий код.

.....

.....

.....

Заточите карандаш
Решение

Завершающий код выполняет уборку и наводит порядок. Для этого диспетчера контекста навести порядок означает подтвердить запись всех данных в базу, после чего закрыть курсор и соединение. Вы должны были добавить код, который, как вы полагаете, понадобится в методе, представленном ниже.

Пока не думайте об этих аргументах.

```
def __exit__(self, exc_type, exc_value, exc_trace) -> None :
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Эти строки записывают в базу данных все значения, присвоенные атрибутам, а также закрывают курсор и соединение. И не забудьте добавить к именам атрибутов приставку «self».

Аннотация указывает, что метод ничего не возвращает. Такая аннотация необязательна, но является правилом хорошего тона.

Диспетчер контекста готов для тестирования

Теперь, закончив метод `__exit__`, можно проверить работу диспетчера контекста и интегрировать его в код веб-приложения. Как обычно, сначала проверим новый код в интерактивной оболочке Python (>>>). Убедитесь, что ваш код полностью соответствует нашему.

Законченный класс диспетчера контекста «UseDatabase».

DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

```
import mysql.connector


class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Ln: 18 Col: 0

«Настоящий» класс будет содержать документацию, но мы убрали ее, чтобы сэкономить место (на странице). Все загружаемые примеры включают комментарии.

Пробная поездка

Откройте файл `DBcm.py` в окне редактирования IDLE и нажмите F5, чтобы приступить к тестированию диспетчера контекста.

Импортируем класс диспетчера контекста из файла модуля «DBcm.py».

Используем диспетчер контекста для отправки некоторого запроса SQL на сервер и получим некоторые данные назад.

Поместим параметры соединения в словарь.

```
Python 3.5.1 Shell
>>>
>>> from DBcm import UseDatabase
>>>
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
>>>
>>> with UseDatabase(dbconfig) as cursor:
        _SQL = """show tables"""
        cursor.execute(_SQL)
        data = cursor.fetchall()


>>> data
[('log',)]
>>>
>>>
                                          Ln: 45  Col: 4
```

Полученные данные могут показаться немного странными... пока вы не вспомните, что «cursor.fetchall» возвращает список кортежей, и каждый из них соответствует записи в наборе результатов (возвращаемом базой данных).

Здесь не так много кода, правда?

Как видите, потребовалось написать совсем немного кода. Мы успешно переместили часть кода для работы с базой данных в класс `UseDatabase`, и теперь инициализация, настройка и завершающие операции выполняются «за кулисами» — диспетчером контекста. Остается только определить параметры соединения и SQL-запрос, который требуется выполнить — все остальное сделает диспетчер контекста. Код, выполняющий настройки и заключительные операции, используется повторно, как часть диспетчера контекста. Кроме того, стало очевиднее основное назначение кода: получение данных из базы и их обработка. Диспетчер контекста скрывает подробности подключения/отключения базы данных (которые не меняются), тем самым позволяя вам сосредоточиться на обработке данных.

Давайте обновим веб-приложение, задействовав диспетчер контекста.

Повторное обсуждение кода веб-приложения, 1 из 2

Мы уже давно не рассматривали код нашего веб-приложения.

В последний раз (в главе 7) мы изменили функцию `log_request`, реализовав сохранение веб-запросов в базу данных MySQL. После этого мы ушли в изучение классов (в главе 8), чтобы найти лучший способ повторного использования кода для работы с базой данных, который мы добавили в `log_request`. Сейчас мы знаем, что лучшее решение (в этой ситуации) — использовать только что написанный класс диспетчера контекста `UseDatabase`.

Код веб-приложения хранится в файле «vsearch4web.py», в папке «webapp».

Помимо изменения функции `log_request`, где нужно задействовать диспетчер контекста, нужно изменить еще одну функцию, `view_the_log`, переключив ее на использование базы данных (сейчас она работает с текстовым файлом `vsearch.log`). Прежде чем приступить к исправлению обеих функций, давайте вспомним (на этой и следующей страницах), как выглядит код веб-приложения. Мы выделили участки кода, требующие изменения.

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

import mysql.connector

app = Flask(__name__)


def log_request(req: 'flask_request', res: str) -> None:
    """Журналирует веб-запрос и возвращаемые результаты."""
    dbconfig = {'host': '127.0.0.1',
                'user': 'vsearch',
                'password': 'vsearchpasswd',
                'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Вместо этого нужно импортировать «DBcm».

Этот код нужно изменить так, чтобы он использовал диспетчер контекста «UseDatabase».

Повторное обсуждение кода веб-приложения, 2 из 2

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    """Извлекает данные из запроса; выполняет поиск; возвращает результаты."""
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    """Выводит HTML-форму."""
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
def view_the_log() -> 'html':
    """Выводит содержимое файла журнала в виде HTML-таблицы."""
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


if __name__ == '__main__':
    app.run(debug=True)
```

Код нужно изменить, чтобы он использовал данные из базы данных при помощи диспетчера контекста «UseDatabase».

Вспомним функцию «log_request»

Начиная изменять функцию `log_request`, чтобы задействовать диспетчер контекста `UseDatabase`, нужно учесть, что большая часть работы уже выполнена (как мы показывали ранее).

Взгляните на `log_request`. В данный момент словарь с параметрами подключения к базе данных (`dbconfig`) определяется внутри `log_request`. Так как этот словарь понадобится еще в одной функции, которую тоже надо изменить (`view_ the_ log`), давайте вынесем его за пределы функции `log_request`, чтобы использовать словарь в других функциях.

Переместим словарь за пределы функции, чтобы при необходимости использовать его в других функциях.

```
def log_request(req: 'flask_request', res: str) -> None:

    dbconfig = {'host': '127.0.0.1',
                'user': 'vsearch',
                'password': 'vsearchpasswd',
                'database': 'vsearchlogDB', }

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Вместо перемещения `dbconfig` в глобальное пространство веб-приложения, было бы полезнее каким-то образом добавить его во внутреннюю конфигурацию.

По счастливому стечению обстоятельств, Flask (подобно многим другим веб-фреймворкам) имеет встроенный механизм конфигурации. Это словарь (который в Flask называется `app.config`), который позволяет устанавливать внутренние настройки веб-приложения. Так как `app.config` — это обычный словарь, мы можем добавлять в него собственные ключи и значения, представляющие данные из `dbconfig`.

Давайте внесем эти изменения прямо сейчас.

Изменение функции «log_request»

После внесения изменений код веб-приложения выглядит так.

Заменили старую инструкцию «import» на новую.

Добавили словарь с параметрами соединения в конфигурацию веб-приложения.

Исправили код, использовав диспетчер контекста «UseDatabase», которому передали настройки из «app.config».

vsearch4web.py - /Users/paul/Desktop/_NewBook/ch09/webapp/vsearch4web.py (3.5.1)

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters
from DBcm import UseDatabase

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }


def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""

    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

Ln: 10 Col: 0

Почти в самом начале файла мы заменили инструкцию `import mysql.connector` другой инструкцией `import`, которая импортирует класс `UseDatabase` из модуля `DBcm`. Файл `DBcm.py` уже содержит инструкцию `import mysql.connector`, поэтому мы удалили `import mysql.connector` из файла веб-приложения (чтобы не импортировать модуль `mysql.connector` дважды).

Также мы переместили словарь с параметрами подключения к базе данных в конфигурацию веб-приложения. И изменили код `log_request`, задействовав наш диспетчер контекста.

После проделанной работы с классами и диспетчерами контекста вы должны свободно читать и понимать код, показанный выше.

Теперь перейдем к функции `view_the_log`. Прежде чем перевернуть страницу, убедитесь, что код вашего веб-приложения точно такой, как показано выше.

Вспомним функцию «view_the_log»

Теперь внимательно рассмотрим функцию `view_the_log`, поскольку с тех пор, как мы детально обсуждали ее, прошло очень много времени. Напомним, что текущая версия этой функции извлекает зарегистрированные данные из текстового файла `vsearch.log`, превращает их в список списков (с именем `contents`) и передает его в шаблон с именем `viewlog.html`.

Извлекает каждую строку из файла и преобразует ее в список экранированных элементов, который затем добавляется в конец списка «contents».

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))

    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Обработанные данные из журнала передаются в шаблон для отображения.

Примерно так выглядит результат отображения данных из списка списков `contents` в шаблон `viewlog.html`. Эта функциональность на данный момент доступна в веб-приложении через URL `/viewlog`.

Данные из «contents» отображенные в шаблон. Обратите внимание, что данные («phrase» и «letters») включаются шаблоном в один стольбец.

Изменился не только этот код

Перед тем как как заняться изменением функции `view_the_log`, чтобы задействовать в ней диспетчер контекста, давайте приостановимся и рассмотрим данные, хранящиеся в таблице `log` в базе данных. Тестируя первую версию функции `log_request` в главе 7, мы заходили в консоль MySQL и проверяли сохраненные данные. Вспомним этот сеанс в консоли MySQL, который уже появлялся на страницах книги ранее.

File Edit Window Help Checking our log DB

```
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                  | letters | ip        | browser_string | results              |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker             | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                  | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker             | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+-------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

Журналируемые данные сохранены в таблице базы данных.

Если данные выше сопоставить с данными, хранящимися в настоящий момент в файле `vsearch.log`, станет понятно, что кое-какие операции в функции `view_the_log` стали лишними, потому что сейчас данные хранятся в таблице. Ниже приведен фрагмент данных из файла журнала `vsearch.log`.

```
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'x', 'y'}
```

Данные хранятся в файле журнала «vsearch.log» в виде одной длинной строки.

Часть кода присутствует в текущей версии `view_the_log` только потому, что до этого момента данные сохранялись в журнал `vsearch.log` как коллекция длинных строк (разделенных вертикальными чертами). Такой формат был выбран не просто так, но нам пришлось написать дополнительный код для его обработки.

Совсем иное дело с данными в таблице `log`, так как они «структурированы по умолчанию». Это избавляет от необходимости выполнять дополнительную обработку внутри `view_the_log`: нужно только извлечь данные из таблицы, которые, по счастью, возвращаются как список кортежей (благодаря методу `fetchall` из DB-API).

Помимо этого, значения `phrase` и `letters` в таблице `log` разделены. Если внести небольшое изменение в код шаблона, данные можно будет вывести в пяти колонках (а не в четырех), и наша таблица в браузере станет еще удобнее и проще для чтения.

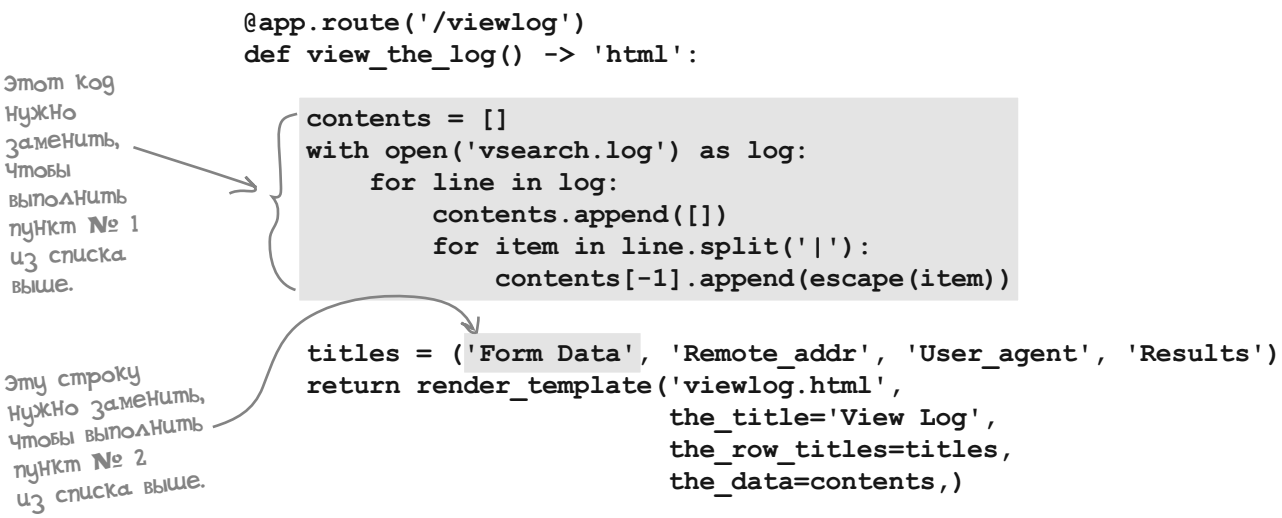
Изменение функции «view_the_log»

Исходя из сказанного на предыдущих страницах, мы должны внести еще два изменения в текущий код `view_the_log`.

1. Извлечь данные из таблицы `log` в базе данных (а не из файла).
2. Настроить список `titles` для поддержки пяти колонок (а не четырех, как было раньше).

Если вы чешете затылок и задаетесь вопросом, почему этот небольшой список исправлений не включает настройку шаблона `viewlog.html`, не удивляйтесь: *этот* файл не требует никаких изменений, так как текущий шаблон способен обработать любое число заголовков и любое количество переданных данных.

Вот так сейчас выглядит код функции `view_the_log`, который мы должны исправить.



SQL-запрос, который нам будет нужен

Ниже показан SQL-запрос для следующего упражнения (где вы измените функцию `view_the_log`), возвращающий все данные из таблицы `log`, сохраненные веб-приложением. Данные из таблицы возвращаются в код на Python в виде списка кортежей. Вы должны будете использовать этот запрос в упражнении на следующей странице.

```
select phrase, letters, ip, browser_string, results
from log
```

Заточите карандаш

Ниже приведена функция `view_the_log`, которую нужно изменить так, чтобы она читала данные из таблицы `log`. Ваша задача — восстановить пропущенный код. Обязательно прочитайте комментарии — они подскажут, что нужно сделать.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    with ..... :

        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""

        .....
        .....

    titles = ( ....., ..... , 'Remote_addr', 'User_agent', 'Results')

    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Используйте здесь диспетчер контекста и не забудьте про курсор.

Отправьте запрос на сервер и извлеките результаты.

Каких заголовков здесь не хватает?

Я только отмечу, что здесь происходит. Новый код получился не только короче, но также проще и понятнее.

Именно. К этому мы и стремились.

Переместив журналируемую информацию в базу данных MySQL, мы избавились от необходимости создавать и обрабатывать текстовый файл.

За счет повторного использования нашего диспетчера контекста мы упростили взаимодействие с MySQL из программного кода на Python. Как это может не понравиться?

Заточите карандаш
Решение

Ниже приведена функция `view_the_log`, которую нужно изменить так, чтобы она читала данные из таблицы `log`. Требовалось восстановить пропущенный код.

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

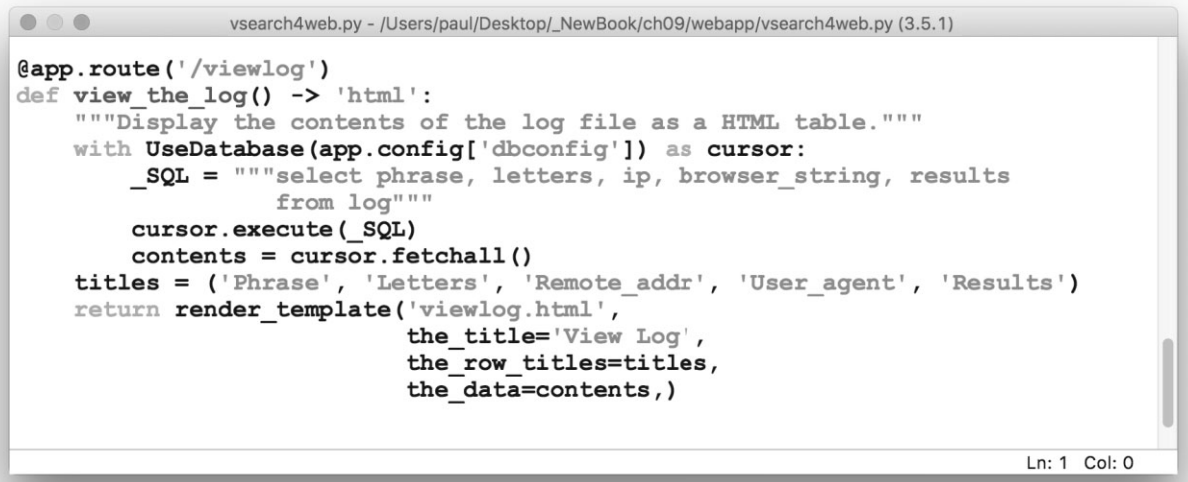
Та же строка кода, что используется в функции «log_request».

Отправляем запрос на сервер, затем извлекаем результаты. Обратите внимание, что извлеченные данные присваиваются переменной «contents».

Добавили соответствующие имена столбцов.

Приближается последняя «Пробная поездка»

Прежде чем опробовать новую версию веб-приложения, потратьте немного времени и убедитесь, что ваша функция `view_the_log` полностью соответствует нашей.



Пробная поездка

Пора испытать веб-приложение, готовое к работе с базой данных.

Убедитесь, что файл `DBcm.py` находится в той же папке, что и файл `vsearch4web.py`, и запустите веб-приложение обычным для вашей операционной системы способом.

- Используйте `python3 vsearch4web.py` в *Linux/Mac OS X*.
- Используйте `py -3 vsearch4web.py` в *Windows*.

Откройте браузер и перейдите на домашнюю страницу вашего веб-приложения (по адресу *http://127.0.0.1:5000*), затем выполните несколько попыток поиска. Убедившись, что поиск работает, откройте страницу с URL */viewlog* и просмотрите содержимое журнала в окне браузера.

Данные для поиска, которые вы вводили, скорее всего будут отличаться от наших, тем не менее вот что мы увидели в окне нашего браузера. Все работает, как ожидалось.

127.0.0.1:5000/viewlog

View Log

Phrase	Letters	Remote_addr	User_agent	Results
life, the universe, and everything	aeiou	127.0.0.1	firefox	{'u', 'e', 'i', 'a'}
hitch-hiker	aeiou	127.0.0.1	safari	{'i', 'e'}
galaxy	xyz	127.0.0.1	chrome	{'y', 'x'}
hitch-hiker	xyz	127.0.0.1	firefox	set()
lightning in a bottle	aeiou	127.0.0.1	firefox	{'i', 'a', 'o', 'e'}
testing the database-enabled webapp	aeiou	127.0.0.1	firefox	{'e', 'a', 'i'}

Страница в окне браузера подтверждает, что при обращении к URL */viewlog* зарегистрированные данные читаются из базы данных MySQL. Это значит, что код в `view_the_log` работает и одновременно подтверждается правильная работа функции `log_request`, которая сохраняет в базе данных результат каждого успешного поиска.

Если хотите, потратьте немного времени и войдите в базу данных MySQL, используя консоль MySQL, чтобы убедиться, что данные благополучно сохранены на сервере базы данных. (Или просто поверьте нам, основываясь на том, что наше веб-приложение отображает их, как показано выше.)

Все что остается...

Пришло время вернуться к вопросам, впервые сформулированным в главе 7.

- *Сколько запросов было обработано?*
- *Какой список букв встречается чаще всего?*
- *С каких IP-адресов приходили запросы?*
- *Какой браузер использовался чаще всего?*

Чтобы ответить на *эти вопросы*, можно просто написать код на Python, но мы не будем делать этого, хотя только что потратили эту и предыдущие две главы, чтобы узнать, как Python и базы данных работают вместе. По нашему мнению, писать код на Python для ответа на такие вопросы — не лучшее решение...

Конечно, лучший способ — это SQL.

Для получения ответов на подобные «вопросы о данных» лучше всего подходит механизм запросов базы данных (в MySQL это SQL). На следующей странице вы увидите, что писать SQL-запросы намного быстрее, чем создавать код на Python.

Знание, когда использовать Python, а когда *нет*, — очень важно, поскольку это знание выделяет Python среди многих других технологий программирования. Классы и объекты поддерживаются многими основными языками, но лишь некоторые имеют в своем арсенале нечто похожее на протокол управления контекстом в Python. (В главе 10 вы узнаете о другой возможности, выделяющей Python среди других языков: декораторах функций.)

Перед тем как перейти к следующей главе, бросим беглый взгляд (всего страничка) на эти SQL-запросы...

Ответы на вопросы о данных

Давайте возьмем вопросы, впервые сформулированные в главе 7, и ответим на каждый с помощью запросов к базе данных, написанных на SQL.

Сколько запросов было обработано?

Если вы хорошо знаете SQL, то этот вопрос вас насмешит. Что может быть проще, чем ответ на него? Вы уже знаете, что самый популярный SQL-запрос — это вывод всех данных из таблицы.

```
select * from log;
```

Чтобы этот запрос вернул количество записей в таблице, надо передать * в SQL-функцию `count`, как показано далее.

```
select count(*) from log;
```

Мы *не* показываем ответы. Если вы хотите их увидеть, запустите запросы в консоли MySQL (см. главу 7, чтобы вспомнить, как это делается).

Какой список букв встречается чаще всего?

SQL-запрос, отвечающий на этот вопрос, выглядит страшновато, но на самом деле в нем нет ничего страшного. Вот он.

```
select count(letters) as 'count', letters
from log
group by letters
order by count desc
limit 1;
```

С каких IP-адресов приходили запросы?

Хорошо знающие SQL здесь, вероятно, подумали: «Это тоже легко».

```
select distinct ip from log;
```

Какой браузер использовался чаще всего?

SQL-запрос, отвечающий на этот вопрос, является незначительной вариацией запроса, который отвечал на второй вопрос.

```
select browser_string, count(browser_string) as 'count'
from log
group by browser_string
order by count desc
limit 1;
```

Итак, вот что имеем: на все гнетущие вопросы найдены ответы с помощью нескольких простых SQL-запросов. Попробуйте выполнить их сами, в консоли `mysql>`, и переходите к следующей главе.

Код из 9-й главы, 1 из 2

Код диспетчера контекста из «DBcm.py».

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Первая половина веб-приложения из «vsearch4web.py».

```
from flask import Flask, render_template, request, escape
from vsearch import search4letters

from DBcm import UseDatabase

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

Код из 9-й главы, 2 из 2

Вторая половина веб-приложения из «vsearch4web.py».

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


if __name__ == '__main__':
    app.run(debug=True)
```

10 декораторы функций

Обертывание функций

Протокол управления контекстом из главы 9 — не единственная возможность улучшить код.

Python также позволяет использовать **декораторы** функций — технологию, с помощью которой можно добавлять код к существующим функциям, *не* изменяя имеющийся код. Если вы увидите в этом сходство с черной магией, не переживайте: ничего подобного. Однако многие программисты на Python считают создание декораторов функций одной из наиболее сложных сторон языка, поэтому пользуются этой возможностью реже, чем должны бы. В этой главе мы покажем, что создавать и использовать декораторы совсем не сложно, несмотря на всю их «продвинутость».

Веб-приложение работает хорошо, но...

Вы показали последнюю версию своего веб-приложения коллегам, и ваша работа их впечатлила. Однако они задали интересный вопрос: *правильно ли позволять любому пользователю просматривать страницу с содержимым журнала?*

Они обратили внимание, что каждый пользователь, знающий о существовании URL */viewlog*, сможет просматривать журнал, вне зависимости от наличия или отсутствия прав на это. Фактически на данный момент все URL нашего веб-приложения являются общедоступными, поэтому любой желающий сможет получить доступ к любому из них.

В зависимости от назначения веб-приложения, это может быть или не быть проблемой. Обычно веб-сайты требуют от пользователя аутентификации, прежде чем показать определенный контент. Возможно, неплохо быть более предусмотрительными, предоставляя доступ по URL */viewlog*. Возникает вопрос: *как ограничить доступ к конкретным страницам в нашем веб-приложении?*

Только аутентифицированные пользователи получают доступ

Чтобы получить доступ к веб-сайту, имеющему ограничения для просмотра контента, обычно требуется ввести идентификатор и пароль. Если ваша комбинация идентификатора/пароля подходит, доступ предоставляется, потому что теперь вы аутентифицированы. После успешной аутентификации система будет знать, что вы имеете право просматривать контент, доступ к которому ограничен. Поддержка этого состояния (аутентифицированный или не аутентифицированный пользователь) кажется простой — как установить `True` (доступ разрешен; вход выполнен) или `False` (доступ запрещен; вход *не* выполнен).

Похоже, это очень просто. Простая HTML-форма может запросить у пользователя его данные, а потом передать серверу булево значение True или False, так ведь?

На самом деле все немного сложнее.

Здесь есть небольшая хитрость (это напрямую следует из особенностей работы веб), которая сделает данную работу чуть-чуть сложнее. Давайте посмотрим, какие имеются сложности (и как их решить), а потом перейдем к решению задачи с разграничением прав доступа.

Веб не имеет состояния

Веб-сервер, в простейшем его виде, не отличается интеллектом: каждый запрос рассматривается им как независимый, не имеющий отношения к предыдущим или последующим запросам.

Это означает, что три запроса, отправленные с вашего компьютера друг за другом, будут рассмотрены веб-сервером как три *независимых* запроса. Даже притом что запросы отправлены из одного и того же браузера, запущенного на одном и том же компьютере, который использует один и тот же IP-адрес (все это веб-сервер видит в запросе).

Как мы и говорили, веб-сервер не отличается интеллектом. Даже если мы считаем, что три запроса, отправленные с нашего компьютера, связаны, веб-сервер вовсе так не считает: *каждый веб-запрос не связан ни с предыдущим, ни с последующим.*

Виноват HTTP...

Причина, почему веб-серверы ведут себя подобным образом, кроется в протоколе, поддерживающем веб, которым пользуются веб-сервер и ваш веб-браузер: HTTP (HyperText Transfer Protocol — протокол передачи гипертекста).

HTTP требует, чтобы веб-серверы работали, как описано выше, и причина в производительности: чем меньше приходится делать веб-серверу, тем больше запросов он сумеет обработать. Высокая производительность достигается за счет отказа от поддержки информации о связях между запросами. Эта информация — известная в HTTP как **состояние** (в данном случае термин не имеет никакого отношения к ООП) — не представляет интереса для веб-сервера, потому что каждый запрос воспринимается им независимо от остальных. Поэтому веб-серверы настроены на быстрый отклик, но **не сохраняют** состояния.

Все это замечательно и здорово, пока не наступает момент, когда веб-приложению нужно что-то запомнить.

Если бы в веб это было так просто.

Когда ваш код выполняется на веб-сервере, его поведение может отличаться от того, что вы видите, когда он работает на вашем компьютере. Давайте разберемся в этом.

Веб-сервер (не ваш компьютер) запускает ваш код

Когда Flask запускает веб-приложение на вашем компьютере, он все время держит его в памяти. А теперь вспомните следующие две строки из нашего веб-приложения, которые мы обсуждали в конце главы 5.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Эта строка НЕ выполняется, когда код импортируется.

Инструкция `if` проверяет, был ли импортирован код или запускается непосредственно (с помощью интерпретатора или инструмента, такого как `PythonAniwhere`). Когда Flask работает на вашем компьютере, код веб-приложения запускается непосредственно, вызовом `app.run`. Однако когда код веб-приложения выполняется веб-сервером, он *импортируется*, и поэтому строка `app.run` **не** выполнится.

Почему? Потому что веб-сервер выполнит код веб-приложения, когда *будет нужно*. Для этого веб-сервер импортирует код приложения, затем вызовет нужные функции, а код веб-приложения все это время будет находиться в памяти. Или веб-сервер может решить загружать/выгружать код веб-приложения по мере необходимости, в периоды простоя веб-сервер будет загружать и выполнять только тот код, который нужен в данный момент. И этот второй режим работы — когда веб-сервер загружает код только при необходимости — может привести к проблемам с сохранением состояния в переменных. Например, подумайте, что произойдет, если добавить в приложение следующую строку.

```
logged_in = False
if __name__ == '__main__':
    app.run(debug=True)
```

Переменная «logged_in» может быть признаком входа пользователя в систему.

Идея в том, чтобы дать другим частям приложения возможность ссылаться на переменную `logged_in` и определять факт аутентификации пользователя. Кроме того, ваш код сможет при необходимости изменять эту переменную (после, скажем, успешного входа в систему). Переменная `logged_in` — *глобальна* по своей природе, она доступна всему веб-приложению для чтения и изменения. Похоже, это вполне разумное решение, но оно имеет *две* проблемы.

Не храните состояния веб-приложения в глобальных переменных.

Во-первых, веб-сервер может выгрузить часть кода веб-приложения в любое время (без предупреждения), поэтому глобальные переменные могут **утратить** присвоенные значения и получить начальные значения, когда код в очередной раз будет импортирован. Если в предыдущий раз функция присвоит переменной `logged_in` значение `True`, при повторном импортировании переменная `logged_in` вновь получит значение `False`, что вызовет проблемы...

Во-вторых, в приложении имеется *единственная копия* глобальной переменной `logged_in`, что нормально только для одного пользователя. Если сразу несколько пользователей попытаются получить доступ к приложению и/или изменить значение переменной `logged_in`, это не только приведет к конфликтам, но и разочарует пользователей. Именно поэтому хранить состояния веб-приложения в глобальных переменных считается плохой идеей.

Самое время поговорить о сеансах

В результате изученного на предыдущей странице нам нужно решить две проблемы.

- Найти способ хранить данные без использования глобальных переменных.
- Найти способ различать данные разных пользователей.

Большинство фреймворков для разработки веб-приложений (включая Flask) решают эти две проблемы с помощью единой технологии: **сеансов**.

Считайте сеанс слоем представления состояния поверх веб, не имеющего состояния.

Добавив небольшой блок идентификационных данных в браузер (*cookie*) и объединив его с небольшим блоком идентификационных данных на веб-сервере (*идентификатор сеанса*), Flask использует технологию работы с сеансами для выполнения нужной нам работы. Вы можете хранить состояние веб-приложения в течение долгого времени, и каждый пользователь веб-приложения получит свою собственную копию данных о состоянии. Ошибки и разочарования в прошлом.

Для демонстрации работы механизма сеансов в Flask рассмотрим маленькое веб-приложение, хранящееся в файле `quick_session.py`. Сначала прочитайте код, обращая внимание на выделенные участки. А затем мы обсудим происходящее.

Готовый код

Код «quick_session.py».

Не забудьте добавить «session» в список импорта.

Секретный ключ должен быть таким, чтобы его было трудно отгадать.

Работа с данными сеанса по мере необходимости.

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'

@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

Поддержка сеансов в Flask позволяет хранить состояние

Чтобы воспользоваться поддержкой сеансов в Flask, нужно в первую очередь импортировать `session` из модуля `flask`, что и происходит в первых строках приложения `quick_session.py`, которое вы только что видели. Считайте `session` глобальным словарем Python, в котором хранится состояние веб-приложения (хотя этот словарь наделен некоторыми суперспособностями).

```
from flask import Flask, session
...
```

Начните с импортирования «session».

Хотя веб-приложение все еще работает в веб, не имеющем состояния, импортирование этого единственного элемента дарит вашему веб-приложению возможность запоминать состояние.

Flask гарантирует сохранность данных в `session` в течение всего времени работы веб-приложения (вне зависимости от того, сколько раз ваш сервер выгружал и загружал ваш код). Кроме того, данные, хранящиеся в `session`, имеют уникальный ключ, соответствующий *cookie* вашего браузера, что гарантирует раздельное хранение данных сеанса для каждого из пользователей.

Узнайте больше о сеансах Flask на http://flask.pocoo.org/docs/0.11/api/#sessions

Как именно Flask делает все это — не важно: главное, что он делает *это*. Чтобы подключить все эти дополнительные блага, нужно инициализировать cookie «секретным ключом», который используется фреймворком Flask для шифрования cookie и его защиты от посторонних глаз. Вот как выглядит все это в `quick_session.py`.

```
...
app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'
...
```

Создать новое приложение Flask обычным способом.

Инициализировать генератор Flask секретным ключом. (Обратите внимание: подойдет любая строка. Хотя, как и любой другой пароль, она должна быть такой, чтобы ее было трудно отгадать.)

Документация с описанием Flask предлагает подбирать такой секретный ключ, чтобы его было трудно отгадать, но, в принципе, здесь можно использовать любую строку. Flask использует ее для шифрования cookie перед отправкой браузеру.

После импортирования `session` и инициализации секретного ключа, словарь `session` можно использовать в своем коде, обращаясь к нему, как к любому другому словарю Python. Если обратиться к веб-приложению `quick_session.py` по адресу URL */setuser* (связанному с функцией `setuser`), оно присвоит введенное пользователем значение ключу `user` в словаре `session`, а затем вернет это значение браузеру.

```
...
@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']
...
```

Значение переменной «user» присваивается ключу «user» в словаре «session».

URL должен содержать значение для присваивания переменной «user» (вы узнаете, как это работает, немного позже).

Сохранив некоторые данные в `session`, рассмотрим, как получить к ним доступ.

Поиск по словарю позволяет получить состояние

Теперь, когда у нас есть значение, связанное с ключом `user` в словаре `session`, мы легко сможем получить его обратно, когда потребуется.

Еще один URL в `quick_session.py`, */getuser*, связан с функцией `getuser`. Эта функция извлекает значение, связанное с ключом `user`, и возвращает его веб-браузеру как часть строкового сообщения. Функция `getuser` показана ниже вместе с проверкой равенства `__name__` и `'__main__'` (см. в конце главы 5).

```
...
@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']

if __name__ == '__main__':
    app.run(debug=True)
```

Как принято во всех приложениях на основе Flask, мы контролируем запуск «app.run» с использованием известных идиом Python.

Доступ к данным в «session» получить несложно. Это поиск по словарю.

Пришло время для Пробной поездки?

Мы практически готовы запустить веб-приложение `quick_session.py`. Однако прежде чем сделать это, подумаем, что именно мы хотим проверить.

Прежде всего, нужно проверить, что приложение сохраняет и извлекает данные о сеансе. Кроме того, хотелось бы убедиться, что несколько пользователей могут одновременно взаимодействовать с приложением, не создавая проблем друг для друга: данные сеанса одного пользователя не должны влиять на данные других пользователей.

Для этого мы сымитируем работу нескольких пользователей, запустив несколько браузеров. Хотя браузеры выполняются на одном и том же компьютере, они устанавливают независимые, индивидуальные соединения: в конце концов, веб не имеет состояния. Если бы мы повторили эти тесты на физически разных компьютерах из трех разных подсетей, результаты были бы теми же, потому что каждый запрос к веб-серверу является изолированным, откуда бы он ни поступал. Вспомним, что `session` — технология создания состояния в Flask поверх веб, не имеющего состояния.

Чтобы запустить это веб-приложение, выполните команду в терминале в *Linux* или *Mac OS X*:

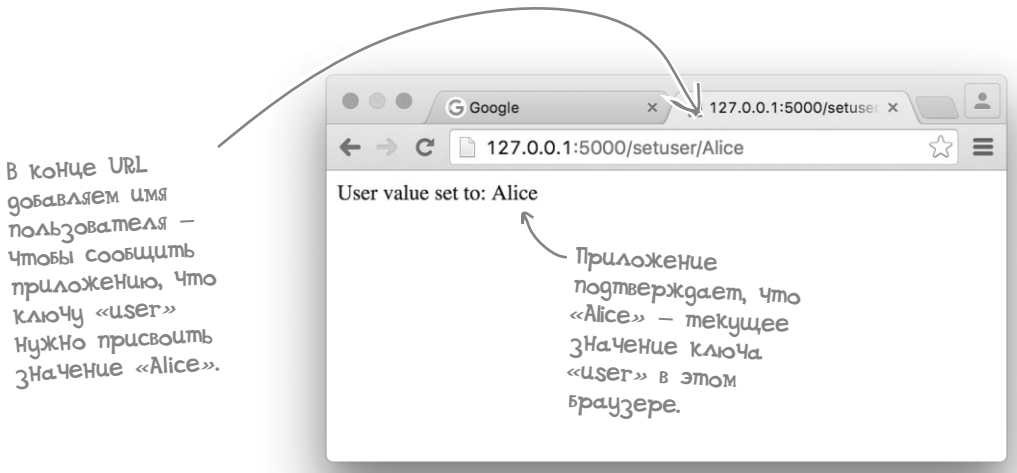
```
$ python3 quick_session.py
```

или команду в консоли *Windows*:

```
C:\> py -3 quick_session.py
```

Пробная поездка, 1 из 2

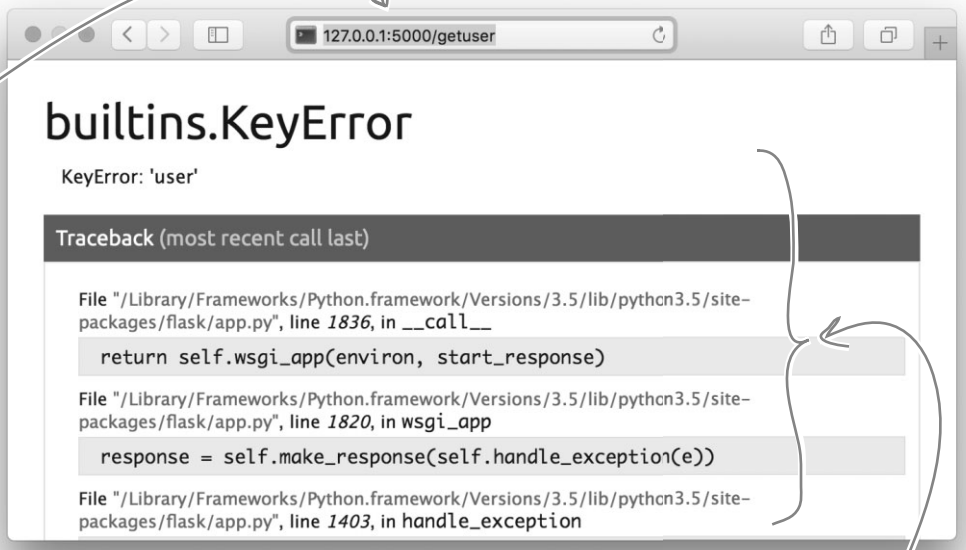
Приложение `quick_session.py` запущено, теперь откроем браузер Chrome и используем его для установки значения ключа `user` в `session`. Для этого наберем */setuser/Alice* в строке адреса, которая сообщит веб-приложению, что ключу `user` нужно присвоить значение `Alice`.



Теперь откроем браузер Opera и используем его, чтобы задать значение `Bob` для `user` (если у вас нет Opera, используйте другой браузер, только не Chrome).

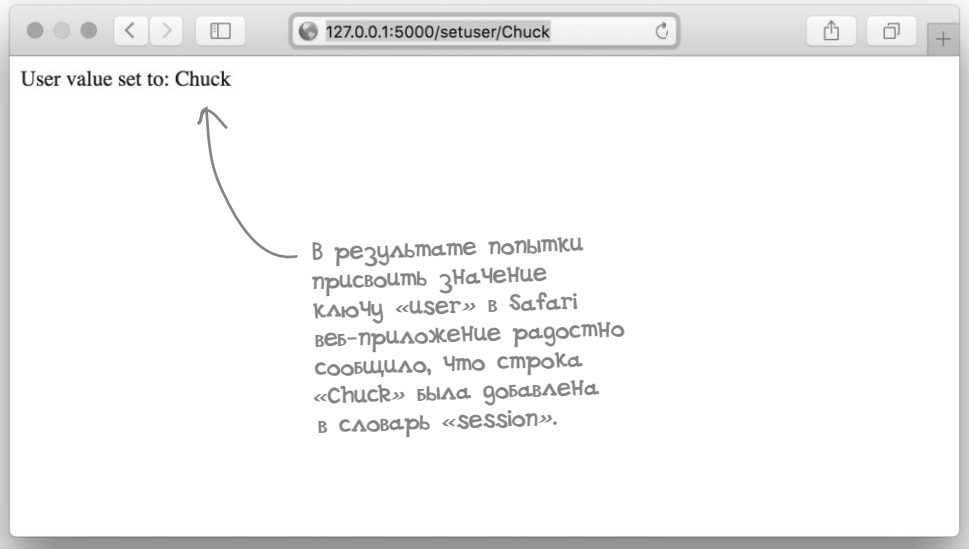
Затем мы открыли Safari (вы можете использовать Edge, если работаете в Windows) и использовали URL */getuser*, чтобы получить текущее значение `user` из нашего приложения. Однако в ответ мы получили внезапное сообщение об ошибке.

URL «/getuser» позволяет проверить текущее значение «user».



Ой! Это же сообщение об ошибке, да? Важная информация содержится в заголовке: мы получили ошибку «KeyError», потому что не использовали Safari для установки значения «user». (Вспомните: мы установили значение «user» в Chrome и Opera, но в Safari этого не сделали.)

Давайте присвоим значение `Chuck` ключу `user` в Safari.

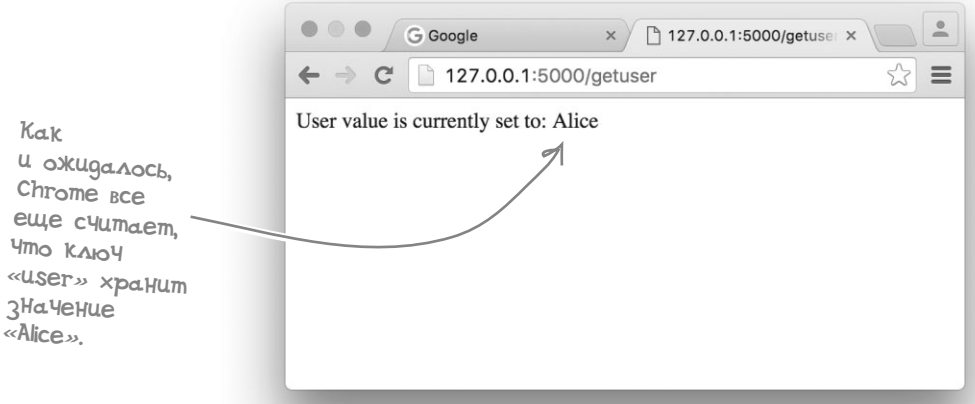


В результате попытки присвоить значение ключу «user» в Safari веб-приложение радостно сообщило, что строка «Chuck» была добавлена в словарь «session».

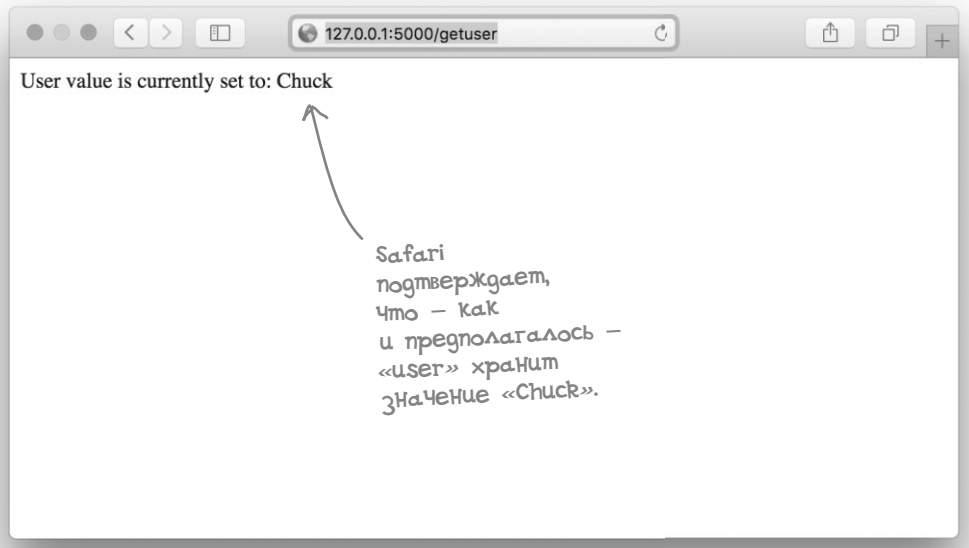
Пробная поездка, 2 из 2

Итак, мы с помощью трех браузеров установили значения для `user`. Теперь убедимся, что веб-приложение (благодаря использованию `session`) не позволит значению `user` в каждом из этих браузеров как-то пересечься с соответствующим значением в остальных браузерах. Мы только что использовали Safari, чтобы присвоить значение `Chuck` ключу `user`. Посмотрим, какое значение вернет нам Opera при обращении к URL */getuser*.

Убедившись, что в Opera для ключа `user` все еще возвращается `Bob`, вернемся в браузер Chrome и проверим URL */getuser*. Как и ожидалось, Chrome подтвердил, что ключ `user` все еще хранит `Alice`.



Мы только что использовали Opera и Chrome для доступа к значению ключа `user` с помощью URL */getuser,* теперь остался Safari. Вот что мы увидели, обратившись по URL */getuser* в Safari и на этот раз не получив сообщения об ошибке, потому что значение `user` теперь установлено (и поэтому нет никаких `KeyError`).



Нет, не совсем — все это делает веб-приложение.

Такое поведение обеспечивается за счет использования словаря `session` в веб-приложении. Автоматически устанавливая уникальный cookie в каждом из браузеров, приложение (благодаря `session`) поддерживает уникальное значение `user` для каждого браузера.

С точки зрения веб-приложения это как бы несколько различных значений `user` в словаре `session` (с уникальными ключами, соответствующими cookie). С точки зрения браузера это как будто бы одно значение `user` (связанное с его собственным, уникальным cookie).

Организация входа в систему с помощью сеансов

Познакомившись с работой `quick_session.py`, мы научились сохранять состояние браузера в словаре `session`. И не важно, сколько браузеров взаимодействует с веб-приложением, данные каждого из браузеров (то есть состояние) сохраняются фреймворком Flask на стороне сервера, благодаря использованию `session`.

Применим новые знания, чтобы решить проблему управления доступом к некоторым страницам в нашем веб-приложении `vsearch4web.py`. Напомним, что мы хотели ограничить доступ пользователей к URL */viewlog*.

Прежде чем начать экспериментировать с рабочей версией кода в `vsearch4web.py`, оставим его на время в покое и поработаем с другим кодом, чтобы в ходе экспериментов понять, что именно нужно сделать. Мы вернемся к коду `vsearch4web.py` чуть позже, когда выберем подходящий способ воплотить наш замысел. И тогда мы изменим код `vsearch4web.py`, чтобы ограничить доступ к */viewlog*.

Вот код другого приложения, основанного на Flask. Как и раньше, потратьте немного времени на чтение кода, прежде чем мы приступим к обсуждению. Вот содержимое файла `simple_webapp.py`.

Готовый код

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
def page3() -> str:
    return 'This is page 3.'

if __name__ == '__main__':
    app.run(debug=True)
```

Это «simple_webapp.py». Вы уже прочли больше половины книги, а значит, сможете прочесть код и понять, что делает это приложение.

Вход в систему

Код `simple_webapp.py` прост: все URL общедоступны, и к ним может обратиться любой, кто умеет пользоваться браузером.

Кроме URL */* по умолчанию (обращение к которому обрабатывается функцией `hello`), есть еще три URL — */page1*, */page2*, */page3* (обращения к которым также обрабатываются соответствующими функциями). Для каждого из URL веб-приложение возвращает свое сообщение.

Это веб-приложение — всего лишь оболочка, но оно подходит для наших целей. Нам нужно, чтобы URL */page1*, */page2*, */page3* были доступны только авторизованным пользователям, а другие не могли их видеть. Для этого мы используем технологию работы с сеансами в Flask.

Начнем с реализации простого URL */login*. Мы не будем беспокоиться о создании HTML-формы, запрашивающей идентификатор пользователя и пароль, а только добавим код, использующий `session` для отражения факта успешной авторизации.

Заточите карандаш

Давайте напишем код для обработки URL */login*. Впишите в пропуски код, который присвоит значение `True` ключу `logged_in` в словаре `session`. Кроме того, функция должна вернуть сообщение «You are now logged in» («Теперь вы в системе»).

Добавьте новый код.

```
@app.route('/login')
def do_login() -> str:
    .....
    return .....
```

Кроме добавления нового кода, обслуживающего URL */login*, нужно внести еще два изменения, чтобы работа с сеансами стала возможной. Уточните, какие изменения нужно внести, по вашему мнению.

1. .....
2. .....

Заточите карандаш
Решение

Вам нужно было написать код для обработки URL */login*. Вы должны были вписать в пропуски код, присваивающий значение `True` ключу `logged_in` в словаре `session`. Кроме того, функция должна была вернуть сообщение «You are now logged in» («Теперь вы в системе»).

Присвоить значение «True» ключу «logged_in» в словаре «session».

```
@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'
```

Вернуть это сообщение браузеру.

Кроме добавления нового кода, обслуживающего URL */login*, нужно внести еще два изменения, чтобы работа с сеансами стала возможна. Вы должны были уточнить, что это за изменения.

1. Нужно добавить 'session' в строку импорта в начале приложения.
2. Нужно установить значение секретного ключа для приложения.

Давайте не забудем это сделать.

Измените код веб-приложения для поддержки входа в систему

Повременим с тестированием кода, пока не добавим URL */logout* и */status*. Прежде чем двигаться дальше, измените свой файл `simple_webapp.py`, включив описанные выше изменения. Обратите внимание: далее приведен не весь код, а только новые фрагменты (выделены фоном).

```
from flask import Flask, session

app = Flask(__name__)

...

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Не забудьте импортировать «session».

Код для обработки URL «/login».

Установите значение для секретного ключа приложения (это позволит использовать сеансы).

Выход из системы и проверка состояния

Наша новая задача — добавить код для обработки адресов URL */logout* и */status*.

Чтобы выйти из системы, можно присвоить значение `False` ключу `logged_in` в словаре `session` или вообще удалить ключ `logged_in` из словаря `session`. Мы реализуем второй вариант; почему — станет понятно, когда напишем код для обработки URL */status*.

Заточите карандаш

Напишем код для обработки URL */logout*, который будет удалять ключ `logged_in` из словаря `session` и возвращать сообщение «You are now logged out» («Вы теперь не в системе»). Впишите в пропуски свой код.

Добавьте свой код для выхода из системы.

Подсказка: если вы забыли, как удалить ключ из словаря, наберите «dir(dict)» в интерактивной оболочке >>> и увидите все методы словаря.

```
@app.route('/logout')
def do_logout() -> str:
    ..... 
    return .....
```

После добавления кода обработки URL */logout* обратите внимание на URL */status*, код обработки которого должен возвращать одно из двух сообщений.

Сообщение «You are currently logged in» («Вы сейчас в системе») возвращается, если ключ `logged_in` существует в словаре `session` (и — по определению — установлен в значение `True`).

Сообщение «You are NOT logged in» («Вы НЕ в системе») возвращается, если в словаре `session` нет ключа `logged_in`. Мы не можем проверить равенство `logged_in` значению `False`, потому что код обработки URL */logout* удаляет ключ из словаря `session`, а не изменяет его значение. (Мы помним, что должны объяснить, почему мы сделали именно так, и мы обязательно сделаем это, но чуть позже. Сейчас просто поверьте, что нужно все сделать именно так.)

Впишите код для URL */status* в пропуски.

Проверьте наличие ключа «logged_in» в словаре «session» и верните соответствующее сообщение.

Добавьте сюда код проверки состояния.

```
@app.route('/status')
def check_status() -> str:
    if .....
        return .....
    return .....
```

Заточите карандаш
Решение

Вам нужно было написать код для обработки URL */logout*, который удаляет ключ `logged_in` из словаря `session` и возвращает сообщение «You are now logged out» («Вы теперь не в системе»).

Используйте метод «pop» для удаления ключа «logged_in» из словаря «session».

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'
```

После добавления кода для обработки URL */logout* нужно было обратить внимание на URL */status*, код обработки которого должен возвращать одно из двух сообщений.

Сообщение «You are currently logged in» («Вы сейчас в системе») должно возвращаться, если ключ `logged_in` существует в словаре `session` (и — по определению — установлен в значение `True`).

Сообщение «You are NOT logged in» («Вы НЕ в системе») должно возвращаться, если в словаре `session` нет ключа `logged_in`.

Нужно было вписать код для обработки URL */status* в пропуски.

Ключ «logged_in» существует в словаре «session»?

Если да — вернуть это сообщение.

Если нет — вернуть это сообщение.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

Еще раз измените код веб-приложения

Мы все еще откладываем тестирование этой версии веб-приложения, но здесь (справа) показана версия кода с выделенными изменениями, которые вы должны внести в свой файл `simple_webapp.py`.

Убедитесь, что ваш код соответствует нашему, прежде чем перейти к «Пробной поездке», которая начнется сразу, как только мы выполним свое обещание.

Две новые процедуры для обработки URL.

```
...

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Почему бы не проверять на ложность?

Когда мы писали код для URL */login*, мы присвоили значение `True` ключу `logged_in` в словаре `session` (это означает, что браузер вошел в систему). Однако в коде для URL */logout* мы не стали присваивать значение `False` ключу `logged_in`, а просто удалили его из словаря `session`. В коде обработки URL */status* мы проверяем состояние, определяя наличие ключа `logged_in` в словаре `session`; мы не проверяем значение ключа `logged_in`. Возникает вопрос: *почему значение* `False` *не используется в приложении как признак выхода из системы?*

Ответ прост, но важен и имеет отношение к особенностям работы словарей в Python. Чтобы проиллюстрировать это, поэкспериментируем в интерактивной оболочке >>> и смоделируем происходящее со словарем `session`, когда он используется в приложении. Обязательно следуйте за нами и внимательно читайте пояснения.

```
Python 3.5.1 Shell
>>>
>>> session = dict()
>>>
>>> if session['logged_in']:
        print('Found it.')


Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    if session['logged_in']:
KeyError: 'logged_in'
>>>
>>> if 'logged_in' in session:
        print('Found it.')


>>> session['logged_in'] = True
>>>
>>> if 'logged_in' in session:
        print('Found it.')


Found it.
>>>
>>> if session['logged_in']:
        print('Found it.')


Found it.
>>>
                                  Ln: 115  Col: 4
```

Создаем новый, пустой словарь «session».

Пытаемся проверить значение «logged_in» с помощью инструкции «if».

Ой! Ключа «logged_in» не существует, поэтому мы получаем «KeyError», и наш код в итоге падает.

Однако если сначала проверить наличие ключа с помощью «in», наш код благополучно выполнится (и не возникнет ошибки «KeyError»), даже если ключа не существует.

Давайте присвоим значение ключу «logged_in».

Проверка на существование с помощью «in» все еще работает, хотя теперь мы получаем положительный результат (потому что ключ существует и ему присвоено значение).

Проверка с помощью «if» работает тоже (потому что ключ существует и имеет значение). Однако если удалить ключ из словаря (с помощью метода «pop»), код снова станет уязвимым для ошибки «KeyError».

Эксперименты показали, что **невозможно** проверить значение ключа, пока пара ключ/значение не существует. Попытка сделать это приводит к появлению ошибки `KeyError`. Поскольку подобных ошибок желательно избегать, код `simple_webapp.py` проверяет наличие ключа `logged_in`, чтобы понять, вошел браузер в систему или нет, а значение ключа не проверяется, что позволяет исключить возможность появления `KeyError`.

Пробная поездка

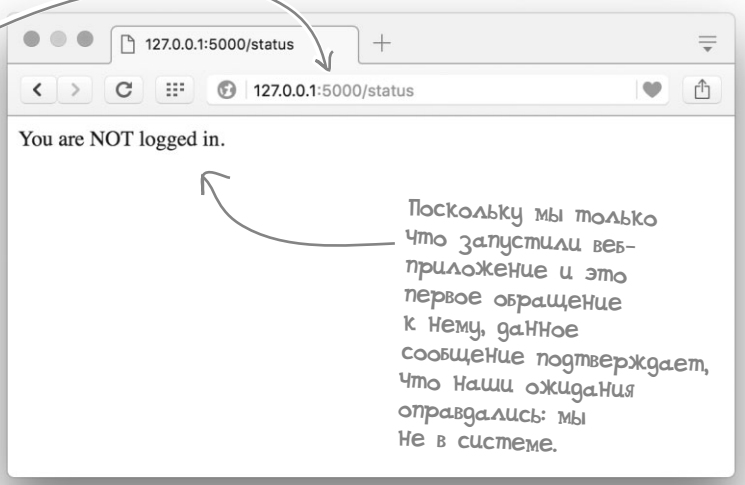
Запустим веб-приложение `simple_webapp.py` и посмотрим, как работают URL */login, /logout* и */status*. Как и в предыдущей «Пробной поездке», мы проверим приложение в нескольких браузерах, чтобы убедиться, что каждый поддерживает собственное состояние «входа в систему». Для начала запустим приложение в командной строке операционной системы.

Для Linux и Mac OS X: **`python3 simple_webapp.py`**

Для Windows: **`py -3 simple_webapp.py`**

Теперь запустим Opera и проверим начальное состояние, обратившись к URL */status*. Как и ожидалось, браузер не в системе.

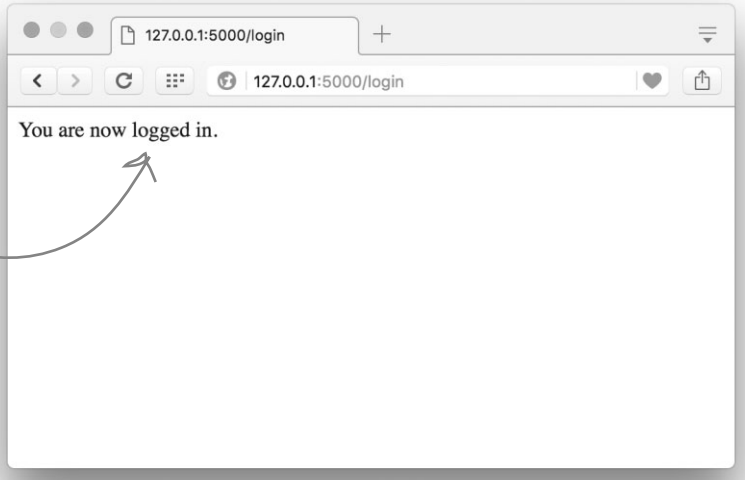
Обратившись к URL «/status», проверим, выполнен ли вход.



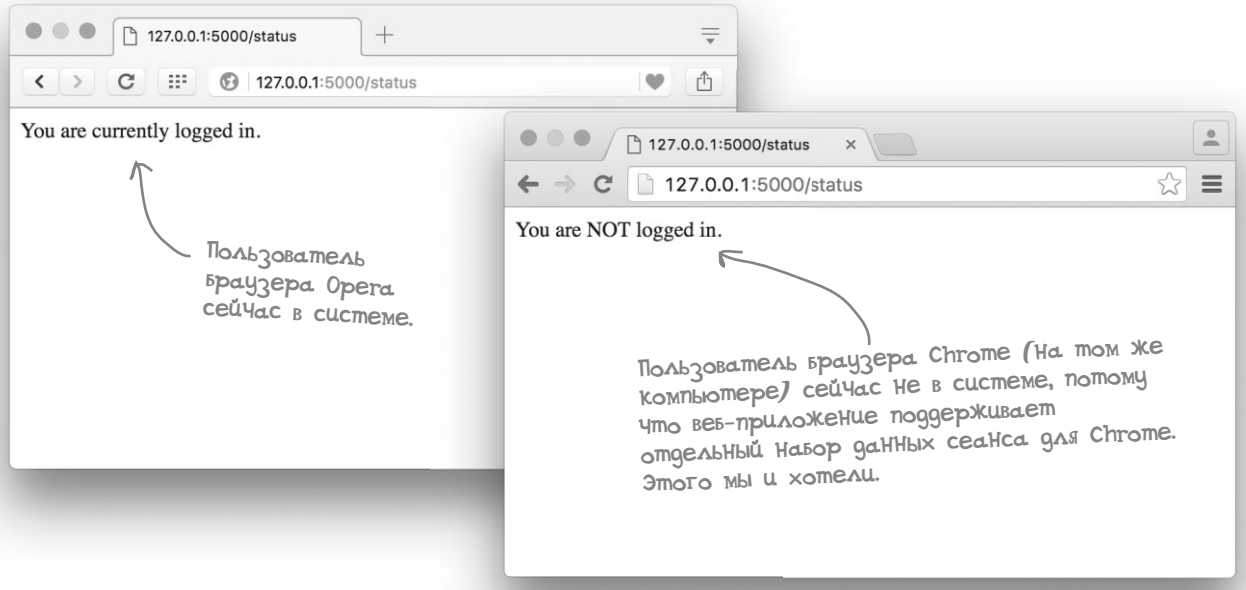
Поскольку мы только что запустили веб-приложение и это первое обращение к нему, данное сообщение подтверждает, что наши ожидания оправдались: мы не в системе.

Давайте сымитируем вход в систему с помощью URL */login*. Это сообщение говорит нам, что мы успешно вошли в систему.

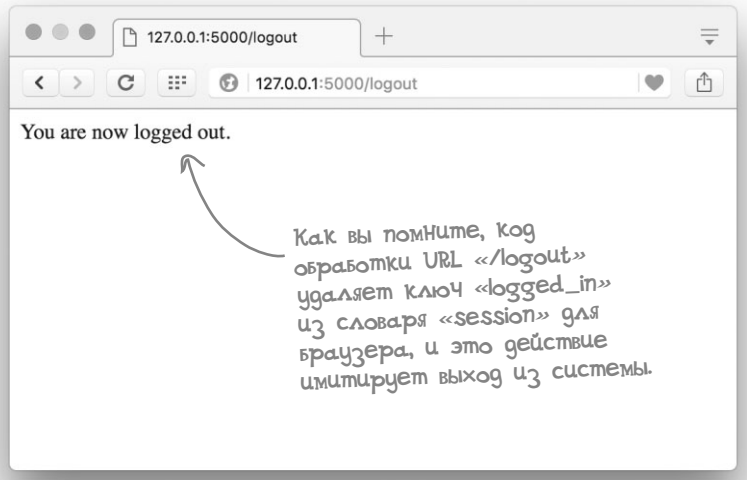
«/login» делает ровно то, что ожидалось. Браузер вошел в систему.



Теперь, когда мы выполнили вход, проверим состояние с помощью URL */status* в Opera. Обращение к этому URL подтверждает, что браузер Opera вошел в систему. Если теперь запустить Chrome и проверить состояние, можно убедиться, что пользователь Chrome не в системе, и это как раз то, что мы хотели получить (потому что каждый пользователь — каждый браузер — должен иметь свое состояние в приложении).



В заключение обратимся к URL */logout* в Opera, чтобы сообщить приложению, что мы выходим.



Хотя мы не просили наших пользователей ввести имя и пароль, URL */login, /logout* и */status* позволяют смоделировать происходящее со словарем `session` в веб-приложении, если бы мы создали HTML-форму для ввода учетных данных пользователя в базе данных. Детали реализации зависят от приложения, но базовый механизм (то есть работа с `session`) всегда один и тот же и не зависит от специфики приложения.

А теперь мы готовы ограничить доступ к */page1, /page2, /page3*?

Можем ли мы теперь ограничить доступ к URL?

Олег. Эй, Денис... на чем застрял?

Денис. Мне нужно подумать, как ограничить доступ к URL */page1*, */page2*, */page3*...

Игорь. Так в чем проблема? У нас уже есть код для работы с функцией, которая обрабатывает обращение к URL */status*...

Денис. И позволяет определить, в системе пользователь или нет, правильно?

Игорь. Да. Нам нужно только скопировать этот код проверки из функции, обрабатывающей */status*, и вставить в функции обработки всех URL, доступ к которым мы хотим ограничить, и все будет отлично!

Олег. Ой, ребята! Копирование и вставка... это настоящая Ахиллесова пята веб-разработчика. Не надо бы копировать и вставлять код таким образом... это может привести к проблемам.

Денис. Конечно! Это же одно из основных правил программирования... я создам функцию с помощью кода из */status*, а затем буду вызывать ее при обработке URL */page1*, */page2*, */page3*. И проблема решена.

Игорь. Мне нравится твоя идея... похоже, это будет работать. (Я знал, что мы не зря посещали эти *скучные* лекции по компьютерным наукам.)

Олег. Подождите... не так быстро. То, что вы предлагаете с функцией, значительно лучше, чем копирование и вставка, но я не уверен, что это лучший способ выполнить задачу.

Денис и **Игорь** (*вместе и удивленно*). Это еще почему??

Олег. Мне кажется, что вы планируете добавить код к функциям, которые обрабатывают */page1*, */page2* и */page3*, а этот код не имеет ничего общего с тем, что делают сами функции. Вы должны только проверить, в системе ли пользователь, прежде чем предоставить доступ к странице, а добавление вызова функции в обработчики всех этих URL мне кажется не совсем верным решением...

Денис. И в чем тогда заключается твоя идея?

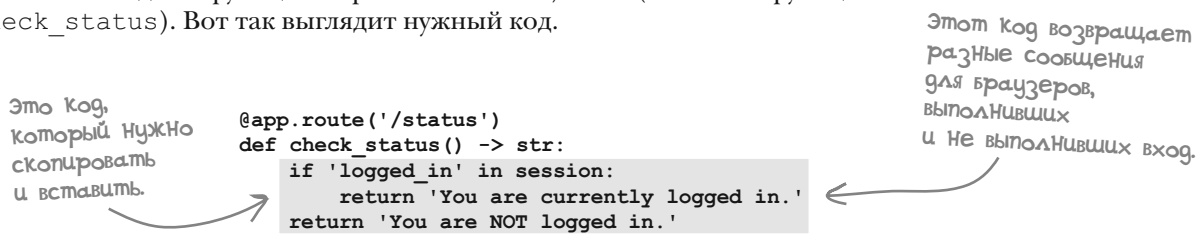
Олег. Я бы создал и использовал декоратор.

Игорь. Ну конечно! Это то что надо! Так и сделаем.

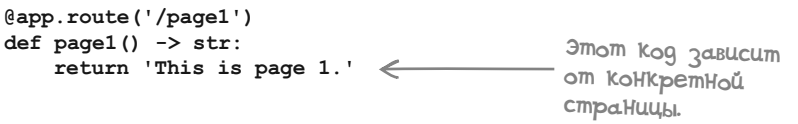
Копирование и вставка редко бывают хорошей идеей

Давайте убедимся, что идеи, предложенные на предыдущей странице, — *не* лучший подход к решению задачи ограничения доступа к определенным страницам.

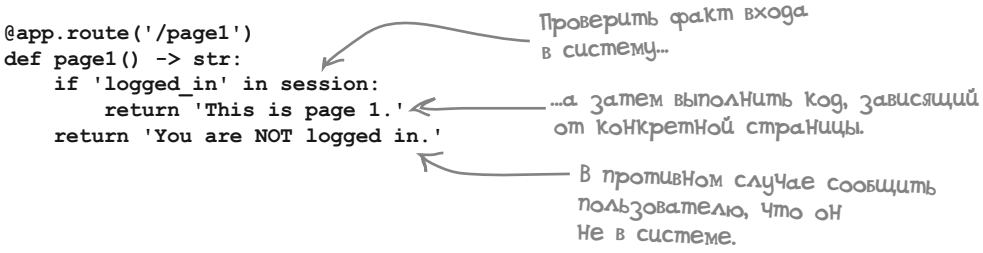
Первое предложение заключалось в копировании и вставке некоторого фрагмента кода из функции-обработчика URL */status* (а именно функции `check_status`). Вот так выглядит нужный код.



Вот как выглядит функция для `page1` сейчас.



Если скопировать выделенный код из `check_status` и вставить в `page1`, результат будет выглядеть так.



Код выше работает, но если постоянно копировать и вставлять фрагменты кода в обработчики других страниц (*/page1*, */page2* и остальных, которые понадобится добавить в веб-приложение), вы создадите настоящий *кошмар для поддержки* в будущем — только представьте, что вы решите изменить способ проверки входа в систему (возможно, проверяя в базе данных идентификатор и пароль, отправленные пользователем).

Помещаем общий код в отдельную функцию

Если есть код, который вы собираетесь использовать в нескольких разных местах, то классическое решение проблемы поддержки, возникающей из-за копирования и вставки, заключается в том, чтобы оформить общий код в виде отдельной функции и вызывать ее, когда понадобится.

Такая стратегия решает проблему поддержки (поскольку общий код присутствует только в одном месте, а не разбросан повсюду). Итак, посмотрим, что даст создание функции входа в нашем веб-приложении.

Создание функции помогает, но...

Теперь создадим новую функцию с именем `check_logged_in`, которая возвращает `True`, если браузер выполнил вход, и `False` — в противном случае.

Это совсем несложно (большая часть кода уже есть в `check_status`); вот как мы написали бы эту новую функцию.

```
def check_logged_in() -> bool:
    if 'logged_in' in session:
        return True
    return False
```

Вместо сообщения этот код возвращает булево значение, соответствующее состоянию входа.

Покончив с функцией, используем ее в функции `page1` вместо копирования и вставки кода:

```
@app.route('/page1')
def page1() -> str:
    if not check_logged_in():
        return 'You are NOT logged in.'
    return 'This is page 1.'
```

Вызываем функцию «check_logged_in», чтобы определить факт входа, затем действуем соответственно.

Тут мы проверяем условие: *не* в системе.

Этот код выполняется, только если браузер в системе.

Эта стратегия лучше, чем копирование и вставка, и теперь можно изменить способ входа в систему, просто внеся изменения в функцию `check_logged_in`. Однако чтобы использовать функцию `check_logged_in`, все еще нужно внести однотипные изменения в код функций `page2` и `page3` (и всех других обработчиков URL, которые вы создадите), и все еще потребуется копировать и вставлять новый код из `page1` в другие функции... Фактически, если сравнить сделанное в функции `page1` на этой странице с тем, что мы делали с этой же функцией на предыдущей странице, можно заметить, что ничего не изменилось, мы делаем *то же самое* — копируем и вставляем! Кроме того, в *обоих* случаях добавленный код **мешает** понять, что на самом деле выполняет `page1`.

Было бы неплохо иметь возможность проверить факт входа пользователя, *не* изменяя существующий код (и не усложняя его). В таком случае каждая функция в веб-приложении будет содержать только код, *непосредственно* отражающий ее назначение, а код проверки состояния входа в систему не будет мешаться. Но есть ли такой способ?

Как сказали наши друзья-разработчики — Игорь, Денис и Олег — несколькими страницами раньше, в языке Python имеется особенность, которая нам поможет: **декоратор**. Декоратор позволяет добавить в существующую функцию дополнительный код и изменить поведение существующей функции *без изменения* ее кода.

Если вы прочитали это предложение и переспросили: *«Что?!?!?»*, не беспокойтесь: это только в первый раз кажется странным. Но как вообще можно изменить поведение функции, не изменяя код? И стоит ли пытаться?

Давайте выясним это, познакомившись с декораторами.

Все это время вы использовали декораторы

Вы уже *использовали* декораторы, пока писали веб-приложение с помощью Flask начиная с 05.

Вот последняя версия `hello_flask.py` из той главы, где выделен пример использования декоратора `@app.route`, который входит в состав Flask. Декоратор `@app.route` применяется к существующей функции (в нашем коде это `hello`) и дополняет следующую за ним функцию `hello`, организуя ее вызов для обработки URL / веб-приложением. Декораторы легко узнать по начальному символу `@`:

Вот декоратор, который — как и все декораторы — начинается с символа @.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Обратите внимание, что вы, как пользователь декоратора `@app.route`, не имеете ни малейшего представления, как именно декоратор творит свое волшебство. Вас волнует только, чтобы декоратор выполнял свои обязанности: связывал функцию с заданным URL. Все подробности работы декоратора скрыты от вас.

Если вы решили создать декоратор, вам нужно вникнуть в детали его работы (как в случае с диспетчером контекста из предыдущей главы) и использовать механизм создания декораторов в Python. Чтобы написать декоратор, вы должны знать и понимать:

1. **Как создать функцию.**
2. **Как передать функцию в аргументе другой функции.**
3. **Как вернуть функцию из функции.**
4. **Как обработать в функции произвольное количество аргументов любого типа.**

Вы уже с успехом создаете и используете свои собственные функции начиная с главы 4, что означает, что список из четырех пунктов сокращается до списка из трех пунктов. Давайте потратим какое-то время для работы над пунктами со 2-го по 4-й, чтобы научиться создавать свои собственные декораторы.

Передаем функцию в функцию

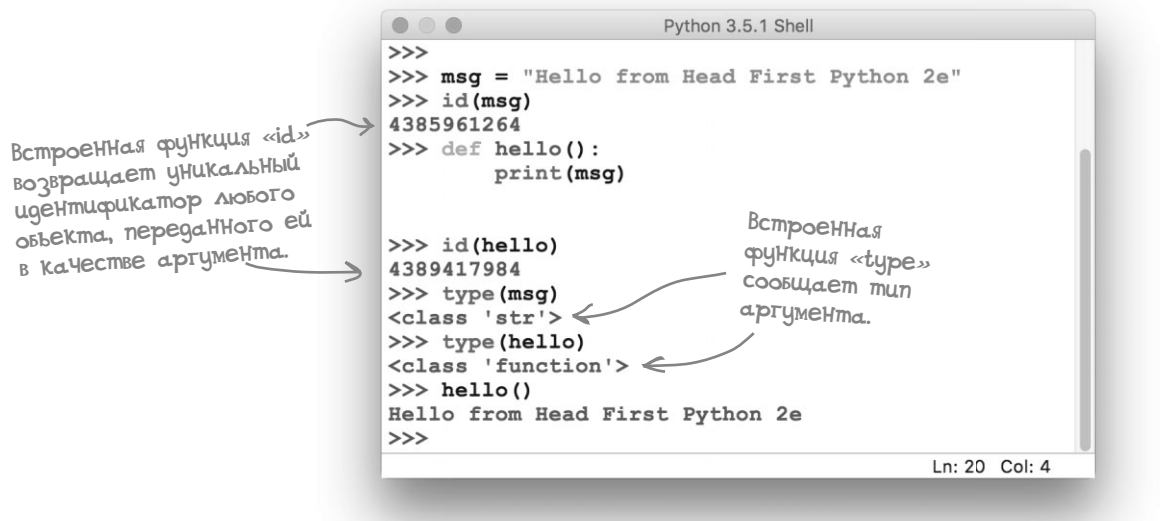
- ☐ Передать функцию в функцию.
- ☐ Вернуть функцию из функции.
- ☐ Обработать аргументы в любом количестве/любого типа.

Мы будем ставить отметки по мере изучения каждого пункта списка.

В главе 2 мы говорили, что в Python *все является объектом*. Хотя это немного странно, но понятие «все» включает и функции, а это означает, что функции — тоже объекты.

Очевидно, что когда функция вызывается, она выполняется. Однако, как и все в Python, функции — это объекты и имеют идентификатор объекта: рассматривайте функции как «объекты-функции».

Рассмотрим короткий сеанс в IDLE, представленный ниже. Переменной `msg` присваивается строка, а затем выводится идентификатор объекта этой переменной с помощью встроенной функции `id`. Далее определяется короткая функция `hello`. Затем она передается в функцию `id`, которая выводит ее идентификатор объекта. Последующие вызовы встроенной функции `type` подтверждают, что `msg` — это строка, а `hello` — функция. В заключение вызывается функция `hello` и на экран выводится текущее значение `msg`.



Мы не хотели заострять ваше внимание на этом прежде, чем вы посмотрите этот сеанс в IDLE, но... вы обратили внимание, *как* мы передали `hello` в функции `id` и `type`? Мы не вызывали `hello`; мы передали ее *имя* каждой из функций в качестве аргумента. Так мы передали функцию в функцию.

Функции могут принимать функции в аргументах

Вызовы функций `id` и `type` в примере выше показывают, что некоторые встроенные функции в Python способны принимать функции в аргументах (точнее *объекты функций*). То, что функция делает со своими аргументами, — внутренние заботы функции. Ни `id`, ни `type` не вызывают полученную функцию, хотя могут это сделать. Давайте посмотрим, как это работает.

Вызываем переданную функцию

- [ ] Передать функцию в функцию.
- [ ] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Когда объект-функция передается другой функции в аргументе, принимающая функция может *вызвать* переданную функцию.

Вот небольшая функция `apply`, которая принимает два аргумента: объект функции и значение. Функция `apply` вызывает полученную функцию, передает ей `value` в качестве аргумента и возвращает результат вызывающему коду.

Функция «apply» принимает объект-функцию в аргументе. Аннотация «object» помогает объяснить назначение аргумента (а использование имени «func» для аргумента — общепринятое соглашение).

func.py - /Users/paul/Documents/func.py (3.5.1)

```
def apply(func: object, value: object) -> object:
    return func(value)
```

Ln: 6 Col: 0

Во втором аргументе функции можно передать любое значение (любого типа). И снова аннотация помогает объяснить, какой именно тип ожидает функция: любой объект.

Вызывается функция (переданная в аргументе), и ей передается единственный аргумент «value». Результат вызова функции возвращается из функции «apply».

Обратите внимание, как аннотации в функции `apply` подсказывают, что она принимает любой объект-функцию вместе с любым другим значением, а затем возвращает какой-то результат (и это все довольно *гибко*). Небольшой тест `apply` в интерактивной оболочке подтверждает, что `apply` работает так, как ожидалось.

Функция «apply» запускает встроенные функции с некоторыми значениями (и работает так, как ожидалось).

Python 3.5.1 Shell

```
>>>
>>> apply(print, 42)
42
>>> apply(id, 42)
4297539264
>>> apply(type, 42)
<class 'int'>
>>> apply(len, 'Marvin')
6
>>> apply(type, apply)
<class 'function'>
>>>
```

Ln: 110 Col: 4

Функция «apply» принимает любой объект в аргументе «value». В этом примере она получает в аргументе «value» саму себя и подтверждает, что это функция.

В каждом из этих примеров первый аргумент функции «apply» присваивается параметру «func».

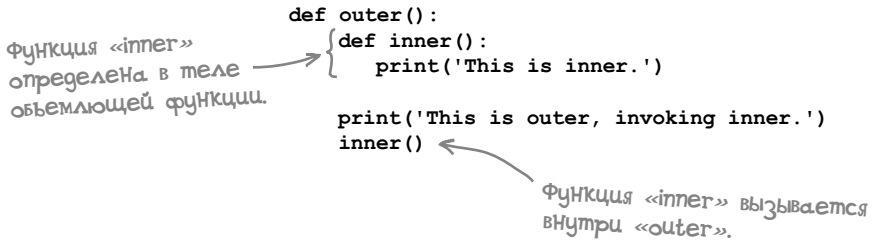
Если вы читаете и удивляетесь, зачем это вообще нужно, не переживайте: все это пригодится, когда мы будем писать декоратор. Пока постарайтесь понять: функции можно передать объект-функцию в аргументе, а затем вызвать его.

Функции можно объявлять внутри функции

- [x] Передать функцию в функцию.
- [ ] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Обычно, создавая функцию, вы берете существующий код и делаете его пригодным для повторного использования, давая ему имя, а затем используете его в качестве тела функции. Это один из наиболее распространенных способов создания функций. Однако, что немного удивляет, в Python в качестве тела функции можно использовать *любой* код, в том числе определение другой функции (часто ее называют *вложенной*, или *внутренней* функцией). Что еще больше удивляет — вложенная функция может *возвращаться* из внешней, объемлющей функции; в результате возвращается *объект функции*. Рассмотрим несколько примеров, в которых функции используются подобным способом.

Первый пример демонстрирует, как функция `inner` вкладывается внутрь другой функции `outer`. Функцию `inner` можно вызвать только из тела функции `outer`, потому что `inner` находится внутри области видимости `outer`.



Когда вызывается функция `outer`, начинает выполняться код в ее теле: определение функции `inner`, вызов встроенной функции `print` внутри `outer`, а затем вызов `inner` (и внутри этой функции тоже вызов `print`). Вот что появляется на экране.

Где это можно использовать?

Рассматривая пример, вам, возможно, трудно представить ситуацию, когда может пригодиться создание функции внутри другой функции. Однако если функция сложная и содержит много строк кода, иногда полезно выделить часть кода во вложенную функцию (и сделать внешнюю функцию более читаемой).

Чаще всего этот прием применяется, чтобы вернуть вложенную функцию из внешней функции с помощью инструкции `return`. Именно это поможет нам создать декоратор.

Теперь посмотрим, что происходит, когда функция возвращает функцию.

Возвращаем функцию из функции

- [x] Передать функцию в функцию.
- [ ] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Наш следующий пример похож на предыдущий, но в нем `outer` больше не вызывает `inner`, а просто возвращает ее. Взгляните на код.

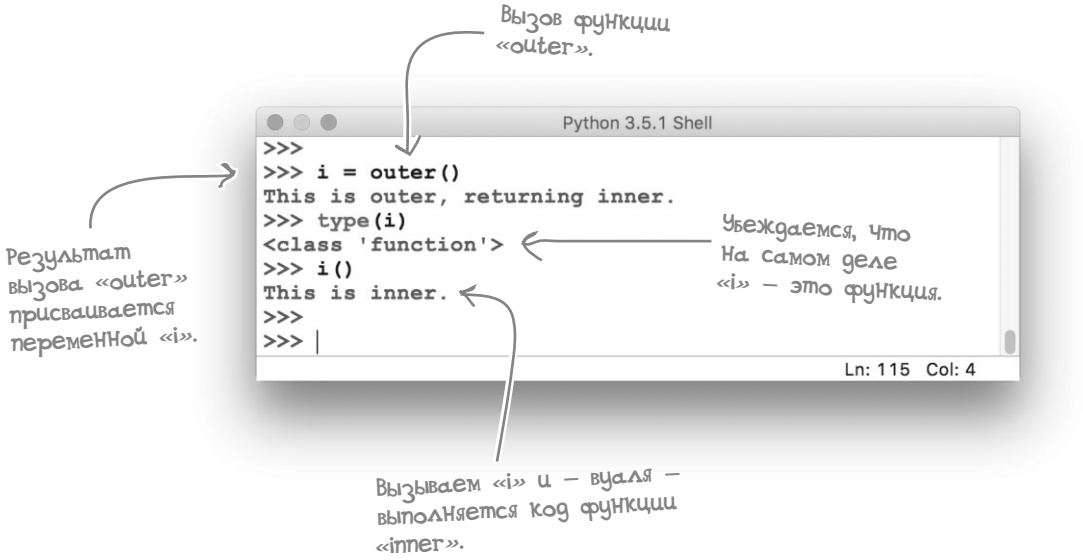
Функция «inner» все еще определена внутри «outer».

```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

Инструкция «return» не вызывает «inner»; она возвращает объект-функцию «inner» вызывающему коду.

Чтобы узнать, что делает новая версия функции `outer`, вернемся в оболочку IDLE и запустим `outer`.

Обратите внимание: в примере мы присваиваем результат вызова `outer` переменной с именем `i`. Затем мы используем `i` в качестве объекта-функции — сначала проверяем ее тип с помощью встроенной функции `type`, а затем вызываем `i` как обычную функцию (добавив в конце скобки). Когда мы вызываем `i`, выполняется функция `inner`. В результате `i` теперь является *псевдонимом* функции `inner`, созданной внутри `outer`.



Пока все в порядке. Теперь мы можем *вернуть* функцию из функции, а также *передать* функцию в функцию. Мы почти готовы собрать все это вместе, чтобы создать декоратор. Но нам еще нужно научиться создавать функции, принимающие любое количество аргументов любого типа. Посмотрим, как это делается.

Принимаем список аргументов

- [x] Передать функцию в функцию.
- [x] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Мы почти все сделали. Еще один вопрос, и мы будем готовы создать декоратор.

Представьте, что вам требуется создать функцию (в примере назовем ее `myfunc`), которую можно вызывать с произвольным количеством аргументов. Например, так.

```
myfunc(10)
```

Один аргумент.

Или так:

```
myfunc()
```

Нет аргументов.

Или как-то так:

```
myfunc(10, 20, 30, 40, 50, 60, 70)
```

Много аргументов (которые в этом примере все являются числами, но могут быть чем угодно: числами, строками, логическими значениями, списками).

Фактически функция `myfunc` может вызываться с *любым* количеством аргументов, главное, что мы заранее не знаем, сколько аргументов будет передано.

Поскольку нельзя определить три разных версии `myfunc`, чтобы обработать эти три разных ситуации, возникает вопрос: *может ли функция принимать произвольное число аргументов?*

Используйте *, чтобы принять произвольный список аргументов

Считайте, что * означает «развернуть в список значений».

Python предоставляет специальную нотацию, которая позволяет указать, что функция может принимать любое количество аргументов (где «любое количество» означает «ноль или больше»). Эта нотация использует символ * для обозначения *любого количества* и в сочетании с именем аргумента (для удобства мы используем `args`) указывает, что функция может принимать список аргументов (хотя технически `*args` — это кортеж).

Вот версия `myfunc`, которая использует эту нотацию для приема любого количества аргументов. Если были переданы какие-либо аргументы, `myfunc` выводит их значения на экран.

Нотация «*args» означает «ноль и более аргументов».

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()
```

Ln: 7 Col: 0

Рассматривайте «args» как список аргументов, который можно обрабатывать подобно любому другому списку (хотя это кортеж).

Значения из списка выводятся в одну строку.

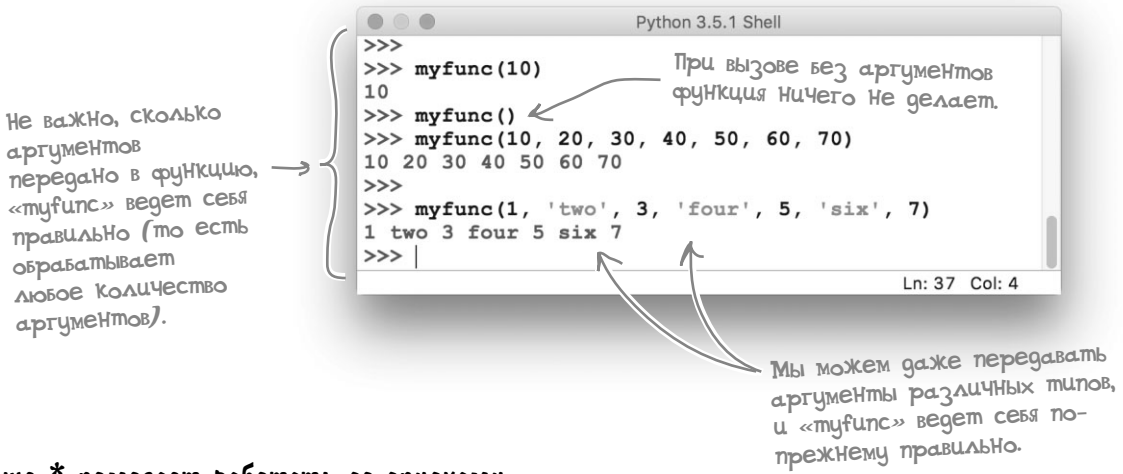
Обработка списка аргументов

- [x] Передать функцию в функцию.
- [x] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Теперь, когда функция `myfunc` создана, посмотрим, можно ли ее вызвать так, как мы планировали на предыдущей странице, а именно:

```
myfunc(10)
myfunc()
myfunc(10, 20, 30, 40, 50, 60, 70)
```

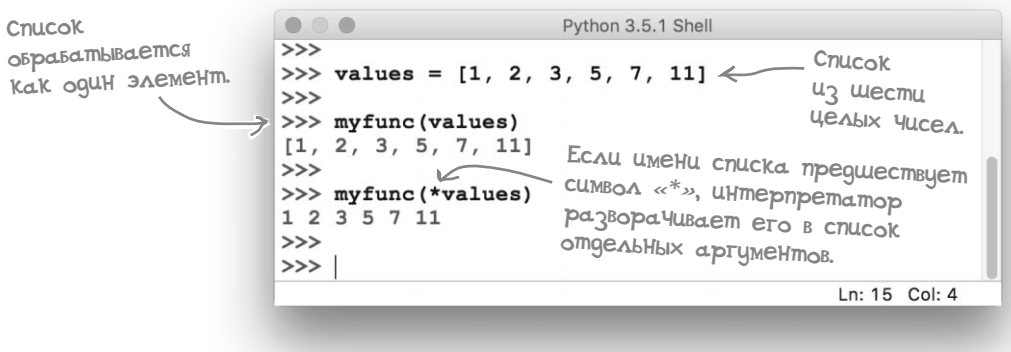
Следующий сеанс IDLE подтверждает, что `myfunc` справляется с задачей. Независимо от количества переданных аргументов (включая *ни одного*), `myfunc` их обрабатывает.



А еще * помогает работать со списками

Если передать в функцию `myfunc` в качестве аргумента список (который может содержать большое количество значений), он будет обработан как один аргумент (потому что это *один* список). Можно потребовать от интерпретатора **развернуть** список, чтобы каждый его элемент обрабатывался как *отдельный* аргумент, добавив перед именем списка символ звездочки (`*`).

Еще один короткий сеанс IDLE демонстрирует отличия при использовании `*`.

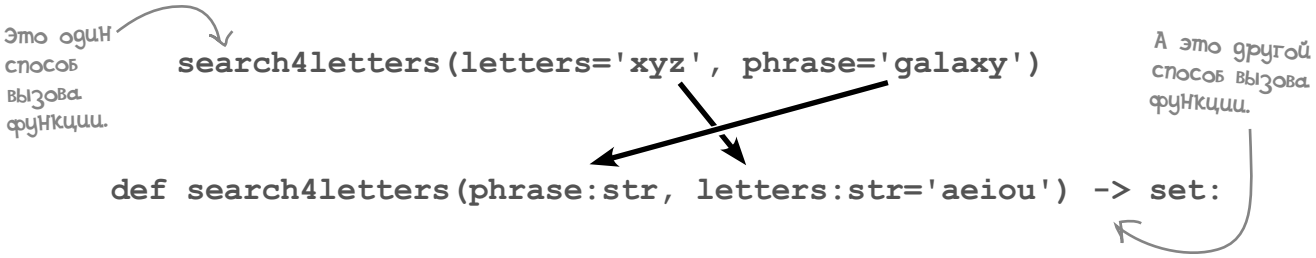


Принимаем словарь аргументов

- [x] Передать функцию в функцию.
- [x] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Передавая значения в функции, можно указывать имена аргументов вместе с их значениями и довериться интерпретатору: он присвоит значения нужным параметрам.

Впервые вы видели этот прием в главе 4, в функции `search4letters`, которая — как вы, наверное, помните — принимала два аргумента: `phrase` и `letters`. Когда используются именованные аргументы, порядок их следования в вызове функции `search4letters` перестает быть важным.



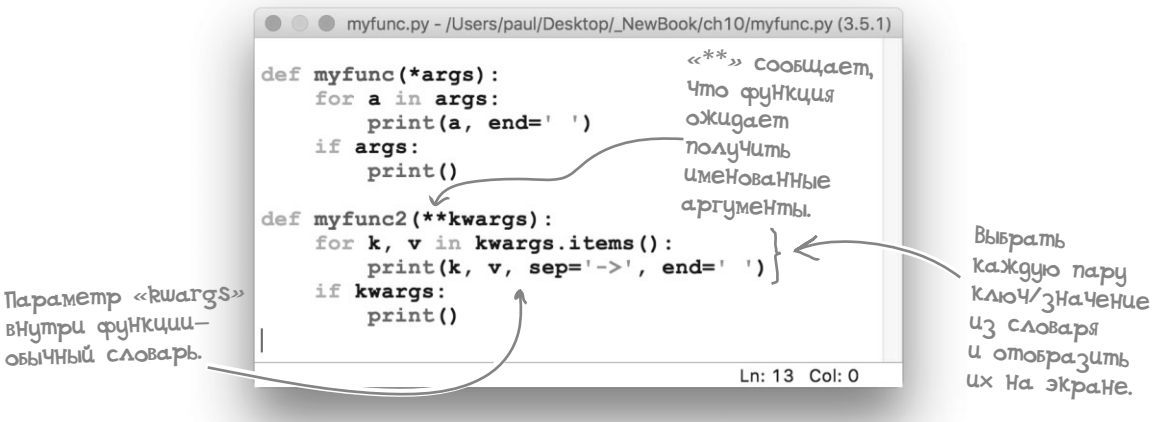
Как и в случае со списками, можно передать функции произвольное количество именованных аргументов, то есть ключей и связанных с ними значений (как в примере с `phrase` и `letters`).

Используйте **, чтобы передать произвольное количество именованных аргументов

Читайте ** как «развернуть словарь ключей и значений».

Кроме нотации `*`, в Python поддерживается нотация `**`, которая позволяет развернуть коллекцию именованных аргументов. В соответствии с соглашениями, для списка аргументов со звездочкой (`*`) используется имя `args`, а для коллекции именованных аргументов с двумя звездочками (`**`) — имя `kwargs`, что означает «keyword arguments» (именованные аргументы). Обратите внимание: вы можете использовать другие имена, кроме `args` и `kwargs`, но очень немногие программисты на Python делают это.

Давайте рассмотрим другую функцию, с именем `myfunc2`, которая принимает произвольное количество именованных аргументов:

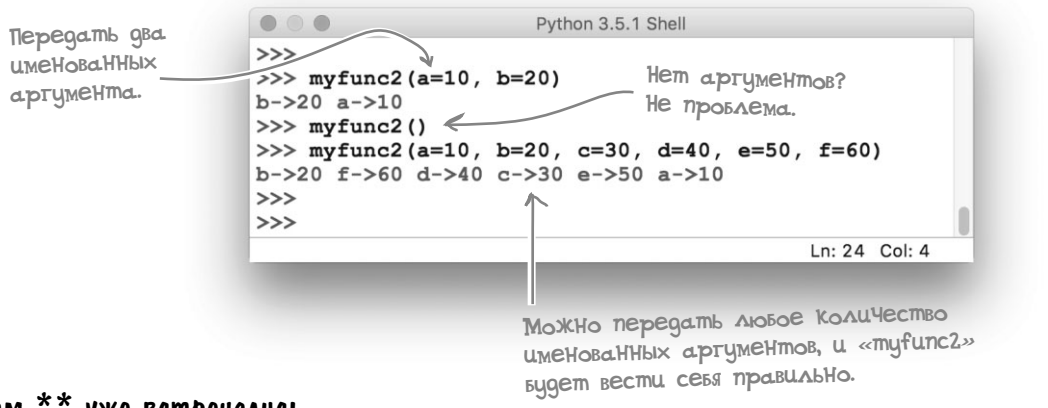


Обработка словаря аргументов

- [x] Передать функцию в функцию.
- [x] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Код в теле функции `myfunc2` принимает словарь аргументов и обрабатывает их, выводя на экран пары ключ/значение.

Следующий сеанс IDLE демонстрирует `myfunc2` в действии. Не важно, сколько пар ключ/значение передано в функцию (возможно, ни одной), `myfunc2` ведет себя правильно.



Нам ** уже встречались

Вы, возможно, уже догадались, к чему все идет, не так ли? Как и в случае с `*args`, при наличии параметра `**kwargs` можно использовать `**` в вызове функции `myfunc2`. Мы не будем показывать, как это работает с `myfunc2`, а просто напомним, что мы уже использовали этот прием ранее. В главе 7, изучая интерфейс DB-API, мы создали словарь для хранения параметров соединения.

Словарь пар ключ/значение.

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }
```

Когда требовалось установить соединение с сервером MySQL (или MariaDB), мы использовали словарь `dbconfig`, как показано ниже. Обратили внимание, как передается аргумент `dbconfig`?

Вам это ничего не напоминает?

```
conn = mysql.connector.connect(**dbconfig)
```

Добавив перед аргументом `dbconfig` символы `**`, мы сообщаем интерпретатору, что аргумент должен использоваться как коллекция ключей и связанных с ними значений. Это равноценно вызову функции `connect` с четырьмя отдельными именованными аргументами.

```
conn = mysql.connector.connect('host'='127.0.0.1', 'user'='vsearch',
                               'password'='vsearchpasswd', 'database'='vsearchlogDB')
```

Принимаем любое количество аргументов любого типа

- [x] Передать функцию в функцию.
- [x] Вернуть функцию из функции.
- [ ] Обработать аргументы в любом количестве/любого типа.

Создавая функции, приятно знать, что Python позволяет принимать список аргументов (с использованием *), а также произвольное количество именованных аргументов (с помощью **). Еще приятнее то, что обе технологии можно сочетать и создавать функции, принимающие любое количество аргументов любого типа.

Вот третья версия `myfunc` (которая имеет очень образное имя `myfunc3`). Функция принимает список аргументов, любое количество именованных аргументов или их комбинацию.

Первоначальная версия «myfunc» работает со списком аргументов.

Функция «myfunc2» работает с любым количеством пар ключ/значение.

Функция «myfunc3» работает с любыми входными данными: списком аргументов, коллекцией пар ключ/значение или их комбинацией.

В строке «def» присутствуют как *args, так и **kwargs.

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()

def myfunc3(*args, **kwargs):
    if args:
        for a in args:
            print(a, end=' ')
        print()
    if kwargs:
        for k, v in kwargs.items():
            print(k, v, sep='->', end=' ')
        print()
```

Ln: 7 Col: 0

Следующий простой сеанс IDLE показывает примеры использования `myfunc3`.

Работает без аргументов.

Работает со списком и именованными аргументами.

Работает со списком.

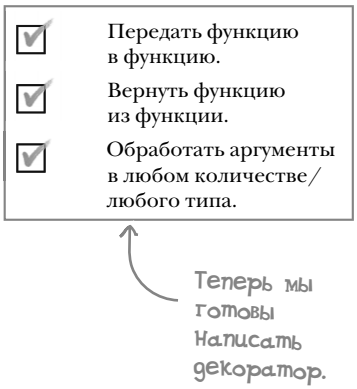
Работает с именованными аргументами.

Python 3.5.1 Shell

```
>>>
>>> myfunc3()
>>> myfunc3(1, 2, 3)
1 2 3
>>> myfunc3(a=10, b=20, c=30)
a->10 b->20 c->30
>>> myfunc3(1, 2, 3, a=10, b=20, c=30)
1 2 3
a->10 b->20 c->30
>>>
```

Ln: 68 Col: 4

Рецепт создания декоратора функции



Три элемента в списке отмечены, и теперь вы изучили все возможности Python, необходимые для создания декоратора. Остается объединить все вместе и создать декоратор.

По аналогии с диспетчером контекста (в главе 9), создание декоратора также осуществляется в соответствии с набором правил, или *рецептом*. Вспомним, что декоратор позволяет добавлять к существующим функциям дополнительный код, не изменяя код самих функций (что, как мы отметили, кажется бредовой идеей).

Чтобы создать декоратор функции, нужно знать следующее.

1. **Декоратор — это функция.**
   С точки зрения интерпретатора декоратор — *всего лишь функция*, хотя она и работает с существующими функциями. Давайте называть эту существующую функцию *декорируемой функцией*. Если вы дочитали до этой страницы, то знаете, что функцию создать просто: нужно использовать ключевое слово `def`.

2. **Декоратор принимает декорируемую функцию как аргумент.**
   Декоратор должен принимать декорируемую функцию как аргумент. Для этого нужно передать декорируемую функцию в виде *объекта функции*. Прочитав предыдущие 10 страниц, вы знаете, что это тоже просто: чтобы получить объект функции, достаточно использовать ее имя без скобок.

3. **Декоратор возвращает новую функцию.**
   Декоратор возвращает новую функцию в качестве возвращаемого значения. Подобно тому как `outer` возвращала `inner` (несколькими страницами ранее), ваш декоратор будет делать что-то подобное, но возвращаемая функция должна *вызывать* декорируемую функцию. Выполнить это — *отважимся ли мы так сказать?* — очень просто, но есть небольшое затруднение, которому посвящен шаг 4.

4. **Декоратор поддерживает сигнатуру декорируемой функции.**
   Декоратор должен убедиться, что функция, которую он возвращает, принимает такое же количество аргументов того же типа, что и декорируемая функция. Количество и типы аргументов какой-либо функции называются **сигнатурой** (потому что строка `def` для каждой функции уникальна).

Пора заточить карандаш, применить эту информацию и создать **наш** первый декоратор.

Итак, нам нужно ограничить доступ к конкретным URL

Мы работаем с кодом `simple_webapp.py`, и нам нужен декоратор, чтобы проверить, был ли выполнен вход. Если вход выполнен, страницы с ограниченным доступом будут доступны. Если вход не выполнен, веб-приложение должно посоветовать пользователю войти в систему, прежде чем просматривать эти страницы. Чтобы решить эту задачу, мы создадим декоратор. Вспомним функцию `check_status`, которая реализует нужную нам логику.

Мы хотим избежать копирования и вставки этого кода.

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

Вспомним: этот код возвращает разные сообщения для браузеров, выполнивших и не выполнивших вход.

Создание декоратора функции

В соответствии с пунктом 1 нашего списка мы должны создать новую функцию. Вспомним...

1. **Декоратор — это функция.**
   С точки зрения интерпретатора декоратор — *всего лишь функция*, хотя она и работает с существующими функциями. Давайте называть эту существующую функцию *декорируемой функцией*. Если вы дочитали до этой страницы, то знаете, что функцию создать просто: нужно использовать ключевое слово `def`.

Из второго пункта следует, что декоратор должен принимать объект функции как аргумент. Вспомним...

2. **Декоратор принимает декорируемую функцию как аргумент.**
   Декоратор должен принимать декорируемую функцию как аргумент. Для этого нужно просто передать декорируемую функцию в виде *объекта функции*. Чтобы получить объект функцию, достаточно использовать ее имя *без* скобок.

Заточите карандаш

Давайте вынесем декоратор в отдельный модуль (чтобы было проще использовать его повторно). Начнем с создания нового файла `checker.py` в текстовом редакторе.

Мы собираемся создать в `checker.py` новый декоратор с именем `check_logged_in`. В пропуски ниже впишите строку `def` для вашего декоратора. Подсказка: используйте `func` в качестве имени аргумента для обозначения объекта функции.

Впишите сюда строку «def» декоратора.

.....

Это не Глупые вопросы

В: **Имеет ли значение, где именно в системе я создам `checker.py`?**

О: Да. Мы планируем импортировать `checker.py` в наше веб-приложение, поэтому нужно гарантировать, что интерпретатор найдет этот код, когда мы добавим строку `import checker`. Пока поместим `checker.py` в ту же папку, что и `simple_webapp.py`.

Заточите карандаш
Решение

Мы решили поместить декоратор в отдельный модуль (чтобы упростить повторное использование).

Мы начали с создания файла `checker.py` в текстовом редакторе.

Новый декоратор (в `checker.py`) получил имя `check_logged_in`, и в пропуск нужно было вписать строку `def` для него.

```
def check_logged_in(func):
```

Декоратор «check_logged_in» принимает единственный аргумент: объект декорируемой функции.

Это было совсем несложно, да?

Вспомним: декоратор — *это просто функция*, которая принимает объект функции как аргумент (`func` в строке `def`).

Перейдем к следующему пункту рецепта создания декоратора, который немного сложнее (но не слишком). Вспомним, что теперь нужно.

3 **Декоратор возвращает новую функцию.**
Декоратор возвращает новую функцию в качестве возвращаемого значения. Подобно тому как `outer` возвращала `inner` (несколькими страницами ранее), ваш декоратор будет делать что-то подобное, но возвращаемая функция должна будет вызывать декорируемую функцию.

Ранее в главе встречалась функция `outer`, которая возвращала функцию `inner`. Вот ее код.

```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

Весь этот код находится в теле функции «outer».

Функция «inner» вложена в «outer».

Объект функции «inner» возвращается как результат вызова «outer». Обратите внимание на отсутствие скобок после имени функции «inner», потому что мы возвращаем объект. Мы *не* вызываем «inner».

Заточите карандаш

Теперь, когда мы написали строку `def` для декоратора, добавим немного кода в его тело. Сделайте следующее.

1. Определите вложенную функцию `wrapper`, которая будет возвращаться из `check_logged_in`. (Здесь можно использовать любое имя функции, но скоро вы увидите, что `wrapper` — подходящий выбор.)

2. Внутрь `wrapper` добавьте код из уже существующей функции `check_status`, которая выбирает одно из двух возможных поведений, в зависимости от состояния входа в систему. Чтобы вам не пришлось листать назад, ниже приведен код `check_status` (важные участки выделены).

```
@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'
```

3. В соответствии с пунктом 3 из рецепта создания декоратора измените код вложенной функции, чтобы он вызывал декорируемую функцию, а не возвращал сообщение «You are currently logged in» («Вы сейчас в системе»).

4. Когда вложенная функция будет написана, верните объект функции из `check_logged_in`.

Добавьте нужный код в тело функции `check_logged_in` в пропуски, оставленные ниже.

```
def check_logged_in(func):
```

1. Определите вложенную функцию.

.....

.....

.....

.....

.....

2 и 3. Добавьте код, который должна выполнять вложенная функция.

4. Не забудьте вернуть вложенную функцию.

Заточите карандаш
Решение

Теперь, когда мы написали строку `def` для декоратора, вам надо было добавить немного кода в его тело. Нужно было сделать следующее.

1. Определить вложенную функцию `wrapper`, которая будет возвращаться из `check_logged_in`.

2. Добавить внутрь `wrapper` код из уже существующей функции `check_status`, которая выбирает одно из двух возможных поведений в зависимости от состояния входа в систему.

3. В соответствии с пунктом 3 из рецепта создания декоратора, изменить код вложенной функции, чтобы он вызывал декорируемую функцию, а не возвращал сообщение «You are currently logged in» («Вы сейчас в системе»).

4. Создав вложенную функцию, вернуть объект функции из `check_logged_in`.

Требовалось добавить нужный код в тело функции `check_logged_in` в пропуски, оставленные ниже.

```
def check_logged_in(func):
    def wrapper():
        if 'logged_in' in session:
            return func()
        return 'You are NOT logged in.'
    return wrapper
```

Вложенная строка «def» означает начало вложенной функции «wrapper».

Если пользователь браузера выполнил вход...

...вызвать декорируемую функцию.

Не забыли вернуть вложенную функцию?

Если пользователь браузера не выполнил вход, вернуть соответствующее сообщение.

Понимаете, почему вложенная функция называется wrapper?

Потратив время на изучение кода декоратора, вы увидите, что вложенная функция не только вызывает декорируемую функцию (хранящуюся в `func`), но и *обертывает* вызов дополнительным кодом. (Имя «wrapper» переводится на русский язык как «обертка». — *Прим. науч. ред.*) В этом случае дополнительный код проверяет наличие ключа `logged_in` в словаре `session`. Если пользователь браузера *не* выполнил вход, декорируемая функция *никогда* не вызывается.

Последний шаг: работа с аргументами

Мы почти закончили — тело декоратора на месте. Осталось убедиться, что декоратор правильно обрабатывает аргументы декорируемой функции, и не важно, какие они. Вспомним пункт 4 нашего рецепта.

4 **Декоратор поддерживает сигнатуру декорируемой функции.** Декоратор должен убедиться, что функция, которую он возвращает, принимает такое же количество аргументов того же типа, что и декорируемая функция.

Когда декоратор применяется к существующей функции, любой ее вызов **заменяется** вызовом функции, которую возвращает декоратор. Как вы видели в решении на предыдущей странице, в соответствии с пунктом 3 рецепта, мы возвращаем обертку функции, которая добавляет новый код. Эта версия *декорирует* существующую функцию.

Но есть еще одна проблема: просто обертки вокруг функции недостаточно, должен поддерживаться *способ вызова* декорируемой функции. Это означает, например, что если декорируемая функция принимает два аргумента, то функция-обертка тоже должна принимать два аргумента. Если бы было заранее известно, сколько потребуется аргументов, можно было бы действовать соответственно. Но, к сожалению, этого нельзя знать заранее, потому что декоратор можно применить к любой существующей функции, которая может иметь — в общем случае — любое количество аргументов любого типа.

Что же делать? Решение должно быть обобщенным, и функция `wrapper` должна поддерживать передачу любого количества аргументов любого типа. Вы знаете, как это сделать, потому что уже видели возможности `*args` и `*kwargs`.

Помните: *args и **kwargs поддерживают любое количество аргументов любого типа.

Заточите карандаш

Изменим функцию `whrapper` так, чтобы она принимала любое количество аргументов любого типа и передавала их в вызов функции `func`. Добавьте в пропуски нужный код.

Что нужно добавить в сигнатуру функции `wrapper`?

```
def check_logged_in(func):
    def wrapper( ..... ):
        if 'logged_in' in session:
            return func( ..... )
        return 'You are NOT logged in.'
    return wrapper
```

Заточите карандаш
Решение

Вам нужно было изменить функцию `wrapper` так, чтобы она принимала любое количество аргументов любого типа и передавала их все в вызов функции `func`.

Использование обобщенной сигнатуры позволяет поддерживать произвольное количество аргументов любого типа. Обратите внимание: мы вызываем «func» с теми же аргументами, которые переданы в функцию «wrapper», независимо от того, какие это аргументы.

```
def check_logged_in(func):
    def wrapper( *args, **kwargs ):
        if 'logged_in' in session:
            return func( *args, **kwargs )
        return 'You are NOT logged in.'
    return wrapper
```

Мы все сделали... да?

Если свериться с рецептом изготовления декораторов, то видно, что мы выполнили все пункты. На самом деле — почти все. Осталось два не решенных момента: один имеет отношение ко всем декораторам, а второй — конкретно к этому.

Сначала рассмотрим конкретный вопрос. Поскольку декоратор `check_logged_in` находится в собственном модуле, нужно убедиться, что весь код, на который он ссылается, импортируется в `checker.py`. Декоратор `check_logged_in` использует словарь `session`, и его нужно импортировать из Flask, чтобы избежать ошибок. Сделать это несложно, достаточно добавить инструкцию `import` в начало модуля `checker.py`.

```
from flask import session
```

Следующий момент касается *всех* декораторов и связан с идентификацией функций интерпретатором. Если при декорировании не взять этот момент на особый контроль, функция может «позабыть» свою идентичность, и это приведет к проблемам. Это происходит по технической причине; в ней мало экзотики, но для ее понимания требуются глубокие знания внутренней кухни Python, лишние для большинства людей. Поэтому в стандартную библиотеку Python был добавлен модуль, принимающий все хлопоты на себя (и вам не нужно об этом беспокоиться). От вас требуется только не забыть импортировать нужный модуль (`functools`) и вызвать единственную функцию `wraps`.

При создании своего декоратора всегда импортируйте, а затем используйте функцию «wraps» из модуля «functools».

В этом есть доля иронии: функция `wraps` реализована как декоратор, поэтому ее нужно не вызывать, а использовать для декорирования функции `wrapper` *внутри* вашего собственного декоратора. Мы уже сделали это за вас, и вы увидите завершенный код декоратора `check_logged_in` на следующей странице.

Наш декоратор во всей красе

Прежде чем продолжить, приведите свой декоратор *в точное* соответствие нашему.

Теперь, когда модуль `checker.py` содержит законченную функцию `check_logged_in`, попробуем использовать ее вместе с `simple_webapp.py`. Вот текущая версия кода веб-приложения (в двух колонках).

```
from flask import Flask, session

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
def page3() -> str:
    return 'This is page 3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'
```

```
@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuess...'

if __name__ == '__main__':
    app.run(debug=True)
```

Вспомните нашу главную цель — ограничить доступ к URL /page1, /page2, /page3, которые на данный момент доступны любому желающему.

Использование декоратора

Изменить код `simple_webapp.py` и использовать в нем декоратор `check_logged_in` совсем несложно. Вот список необходимых изменений.

1. **Импортировать декоратор**
   Декоратор `check_logged_in` нужно импортировать из модуля `checker.py`. Дополнительная инструкция `import` в начале веб-приложения поможет справиться с этой задачей.

2. **Удалить весь ненужный код**
   Теперь, когда имеется декоратор `check_logged_in`, отпала необходимость в функции `check_status`, так что ее можно удалить из `simple_webapp.py`.

3. **Использовать декоратор там, где это нужно**
   Чтобы использовать декоратор `check_logged_in`, нужно передать ему любую из наших функций с помощью синтаксиса `@`.

Ниже приведен код `simple_webapp.py` со всеми изменениями, перечисленными выше. Обратите внимание, что URL */page1*, */page2*, */page3* теперь имеют по два декоратора: `@app.route` (входит в состав Flask) и `@check_logged_in` (который мы только что создали).

Применяем декоратор к существующей функции с помощью синтаксиса @.

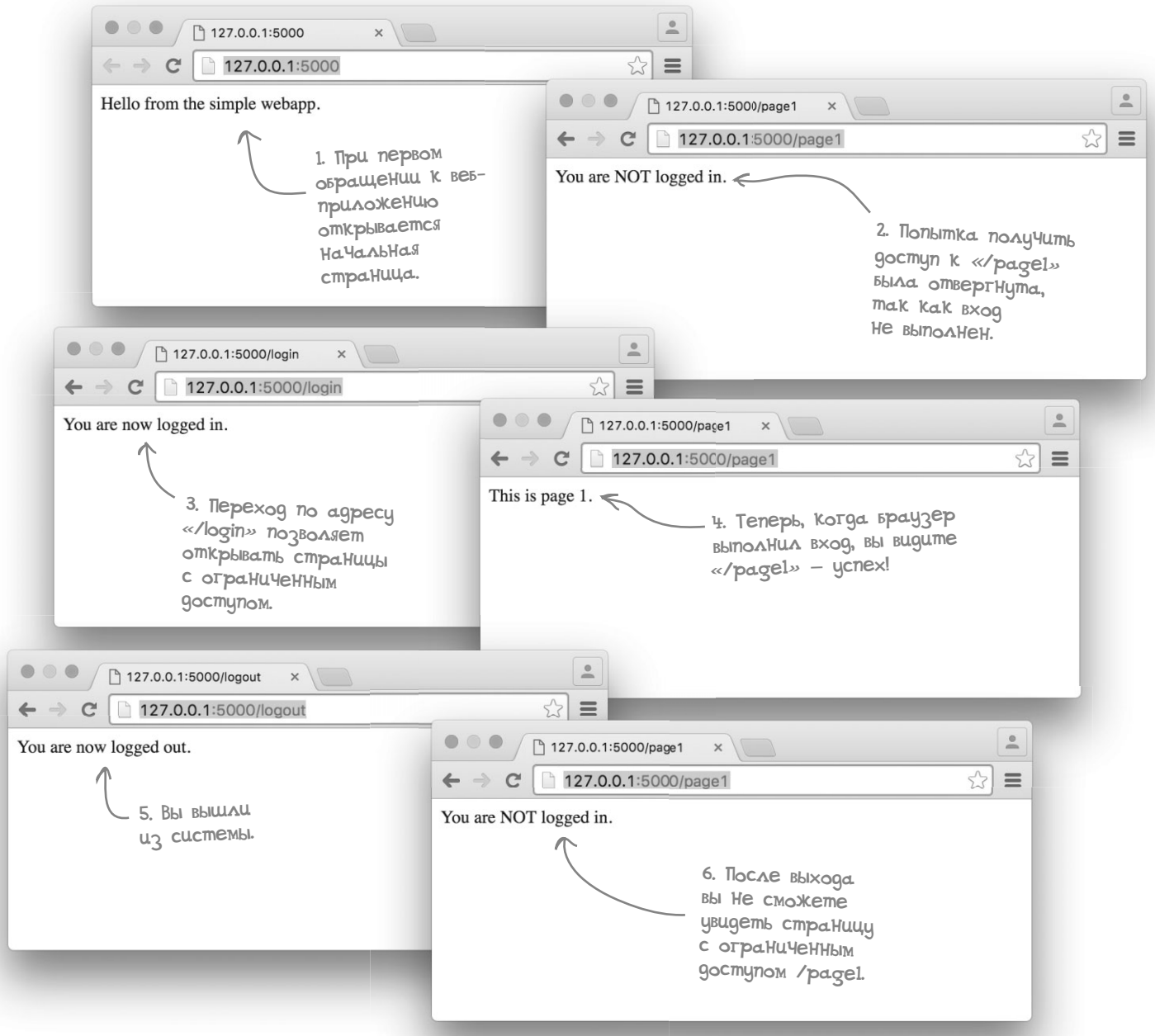
```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

app.secret_key = 'YouWillNeverGuess...'

if __name__ == '__main__':
    app.run(debug=True)
```

Не забудьте внести выделенные изменения, прежде чем продолжить.

Пробная поездка

Чтобы убедиться, что декоратор работает как нужно, протестируем последнюю версию `simple_webapp.py`, в которой он используется.

Запустив веб-приложение, попробуйте открыть в браузере страницу */page1* до того, как выполните вход. После входа в систему попытайтесь снова открыть */page1*, а затем, выполнив выход, попробуйте получить доступ к ограниченному контенту еще раз. Давайте посмотрим, что произошло.



Красота декораторов

Рассмотрим еще раз код нашего декоратора `check_logged_in`. Обратите внимание, как он абстрагирует логику проверки состояния входа, собрав необходимый код (потенциально сложный) в одном месте — *внутри* декоратора — и затем сделав его доступным для всего остального кода, благодаря синтаксису `@check_logged_in`.

checker.py - /Users/paul/Desktop/_NewBook/ch10/checker.py (3.5.1)

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

Ln: 13 Col: 0

Код кажется немного бредовым, но это не так.

Декоратор создает абстракцию, которая позволяет сделать код более читаемым. Посмотрите на использование декоратора для URL */page2*.

simple_webapp4.py - /Users/paul/Desktop/_NewBook/ch10/simple_webapp4.py (3.5.1)

```
@app.route('/page2')
@check_logged_in
def page2():
    return 'This is page 2.'
```

Ln: 1 Col: 0

Использование декоратора делает код более простым для чтения.

Заметьте, код функции `page2` занят только решением своей основной задачи: вывод на экран содержимого страницы */page2*. В примере `page2` состоит из единственной простой инструкции; ее было бы трудно читать и понимать, если бы она содержала *также* проверку состояния входа. Использование декоратора для отделения логики проверки — большое достижение.

Декораторы не бредовые, они классные.

Отделение логики — одна из причин большой популярности декораторов в Python. Вторая, если задуматься, заключается в том, что, создав декоратор `check_logged_in`, мы написали код, который *добавляет к существующей функции дополнительный код, изменяя поведение существующей функции без изменения ее кода.* Когда мы впервые говорили об этом в начале главы, эта идея казалась бредовой. Но теперь, когда вы сделали это, она уже не кажется чем-то особенным, верно?

Создаем еще декораторы

После создания `check_logged_in` вы можете использовать его код в качестве основы для новых декораторов.

Чтобы упростить вам жизнь, далее приведен универсальный шаблон (в файле `tmpl_decorator.py`), который можно использовать как основу для новых декораторов.

tmpl_decorator.py - /Users/paul/Desktop/_NewBook/ch10/tmpl_decorator.py (3.5.1)

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Code to execute BEFORE calling the decorated function.

        # 2. Вызов декорируемой функции и возврат
        #    полученных от нее результатов.
            return func(*args, **kwargs)

        # 3. Код для выполнения ВМЕСТО вызова декорируемой функции.
    return wrapper
```

Ln: 15 Col: 0

Замените эти комментарии кодом вашего нового декоратора.

Этот шаблон можно изменять в соответствии с вашими потребностями. Нужно только дать новому декоратору подходящее имя, а затем заменить три комментария в шаблоне собственным кодом.

Если вашему декоратору нужно вызывать декорируемую функцию без возврата результата, хорошо. В конце концов, код в функции `wrapper` — это ваш код, и вы можете создать его каким пожелаете.

В: Разве декораторы не похожи на диспетчеры контекста из предыдущей главы — оба позволяют добавлять в код новые возможности?

О: Хороший вопрос. Ответ: и да, и нет. Да, декораторы и диспетчеры контекста позволяют добавлять дополнительную логику в существующий код. Но это не одно и то же. Декораторы нацелены на добавление дополнительных возможностей к существующим функциям, а диспетчеры контекста больше направлены на создание определенного контекста выполнения и добавления кода, который должен выполниться перед инструкцией `with`, и кода, который **всегда** должен выполняться после завершения инструкции `with`. Нечто подобное можно сделать с декораторами, но если вы будете так делать, большинство программистов на Python посчитают вас слегка ненормальным. Также обратите внимание, что декоратор не обязан что-либо делать после вызова декорируемой функции (как декоратор `check_logged_in`, который ничего не делает после вызова). Такое поведение декоратора очень отличается от протокола, которого должны придерживаться диспетчеры контекста.

Назад к ограничению доступа к /viewlog

Теперь, создав механизм ограничения доступа к некоторым URL в `simple_webapp.py`, мы легко сможем применить его в другом веб-приложении.

В том числе в `vsearch4web.py`, где требуется ограничить доступ к URL */viewlog*. Для этого достаточно скопировать функции `do_login` и `do_logout` из `simple_webapp.py` в `vsearch4web.py`, импортировать модуль `checker.py`, а затем декорировать функцию `view_the_log` декоратором `check_logged_in`. Возможно, вам захочется сделать функции `do_login` и `do_logout` более сложными (например, сравнить учетные данные, введенные пользователем, с хранящимися в базе данных), но основная работа — связанная с ограничением доступа к конкретным URL — уже выполняется декоратором `check_logged_in`.

Что дальше?

Чтобы не описывать еще раз все, что было сделано в `simple_webapp.py` и на что было потрачено так много времени, но на этот раз с `vsearch4web.py`, мы предлагаем вам изменить код `vsearch4web.py` *самостоятельно*. В начале следующей главы мы покажем обновленную версию `vsearch4web.py`, которую вы сможете сравнить со своей, потому что наш новый код будет обсуждаться в следующей главе.

До настоящего времени весь код в книге был написан в расчете, что ничего плохого не может произойти и ошибки никогда не случаются. Мы сделали это сознательно, потому что хотели научить вас основам программирования на Python, прежде чем переходить к обсуждению таких тем, как обработка ошибок, предотвращение ошибок, обнаружение ошибок, работа с исключениями и тому подобное.

Но теперь мы достигли точки, когда больше не можем следовать этой стратегии. Наш код выполняется в реальном окружении, и все, что в нем происходит или может происходить, иногда приводит к ошибкам. Некоторые ошибки можно предотвратить (или исправить их последствия), некоторые — нет. Желательно, чтобы при любой возможности код обрабатывал ошибочные ситуации и завершал работу аварийно только в тех случаях, которые нам не подконтрольны. В главе 11 мы рассмотрим различные стратегии действий, когда что-то пошло не так.

Но прежде кратко перечислим ключевые моменты этой главы.

КОНТРОЛЬНЫЙ СПИСОК

- Чтобы сохранить состояние на стороне сервера внутри веб-приложения на основе фреймворка Flask, используйте словарь ***session*** (и не забудьте выбрать трудный для угадывания ***secret_key***).
- Функцию можно передать как аргумент другой функции. Имя функции (без скобок) ссылается на **объект функции**, с которым можно работать как с любой другой переменной.
- Получив объект функции в аргументе, ее можно **вызвать**, добавив скобки.
- Функция может быть **вложена** в другую функцию (в таком случае ее область видимости будет ограничена телом внешней функции).
- Функция может не только принять другую функцию в аргументе, но и **вернуть** вложенную функцию как возвращаемое значение.
- **`*args`** означает «список элементов».
- **`**kwargs`** означает «словарь ключей и значений». Читайте «kw» как «keywords» (ключевые слова или ключи).
- Звездочки — ***** и ****** — можно использовать для передачи в функцию списка или коллекции ключей и значений в виде одного (разворачиваемого) аргумента.
- Использование (***`*args, **kwargs`***) в **сигнатуре функции** позволяет создавать функции, принимающие любое количество аргументов любого типа.
- Используя новые особенности языка, представленные в этой главе, вы научились создавать **декораторы функций**, которые изменяют поведение существующих функций, не изменяя их код. Звучит немного бредово, но на самом деле здорово (и очень полезно).

Код из 10-й главы, 1 из 2

Код «quick_session.py».

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'


@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']


@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']


if __name__ == '__main__':
    app.run(debug=True)
```

Это «checker.py», содержащий код декоратора из этой главы: «check_logged_in».

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

Это «tmpl_decorator.py», очень удобный для повторного использования шаблон создания декораторов.

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Код для выполнения ПЕРЕД вызовом декорируемой функции.

        # 2. Вызов декорируемой функции и возврат
        #    полученных от нее результатов.
            return func(*args, **kwargs)

        # 3. Код для выполнения ВМЕСТО вызова декорируемой функции.
    return wrapper
```

Код из 10-й главы, 2 из 2

```
from flask import Flask, session

from checker import check_logged_in

app = Flask(__name__)


@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'


@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'


@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'


@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'


@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'


@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Это веб-приложение «simple_webapp.py», объединяющее весь код из главы. Когда потребуется ограничить доступ к некоторым URL, пользуйтесь стратегией, представленной в нашем приложении.

Мы думаем, что использование декораторов делает код этого веб-приложения более простым для чтения и понимания. А вы так не думаете? ☺

11 обработка исключений

Что делать, когда что-то идет не так

Каким бы хорошим ни был ваш код, иногда все равно что-то идет не так.

Вы успешно выполнили все примеры в книге и, скорее всего, убедились, что имеющийся код работает. Значит ли это, что его можно считать надежным? По всей видимости, нет. Писать код и надеяться, что ничего плохого не случится, в лучшем случае наивно. В худшем случае — опасно, поскольку что-то непредвиденное порой все же происходит (и будет происходить). При написании кода лучше быть осторожным, а не доверчивыми. Осторожным, чтобы ваш код делал именно то, что вам нужно, и правильно реагировал, если что-то пойдет не так. В этой главе вы увидите, что может пойти не так, и узнаете, что делать, когда (и, чаще всего, прежде чем) такое произойдет.

Длинное упражнение

Мы начинаем главу сразу с погружения. Ниже показана последняя версия веб-приложения `vsearch4web.py`. Это — обновленная версия, использующая декоратор `check_logged_in` из предыдущей главы для ограничения доступа к странице с адресом URL */viewlog*.

Прочтите этот код, а затем обведите карандашом и снабдите комментариями те места, где, как вы думаете, могут возникать проблемы при выполнении в рабочем окружении. Выделите все, что, на ваш взгляд, может стать источником любых проблем, а не только потенциальных проблем или ошибок времени выполнения.

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Длинное упражнение Решение

Вам нужно было прочитать код, представленный ниже (который есть не что иное, как обновленная версия веб-приложения `vsearch4web.py`). Затем карандашом обвести и снабдить комментариями те части, где, по вашему мнению, могут возникать проблемы при выполнении в рабочем окружении. Вы должны были выделить все, что, на ваш взгляд, может стать источником любых проблем, а не только потенциальных проблем или ошибок времени выполнения. (Мы пронумеровали аннотации, чтобы на них было проще ссылаться.)

```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

1. Что случится, если попытка подключения к базе данных потерпит неудачу?

2. Эти SQL-запросы защищены от таких неприятностей, как атаки вида инъекция кода SQL или межсайтовый скриптинг?

3. Что произойдет, если выполнение этих SQL-запросов потребует много времени?

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

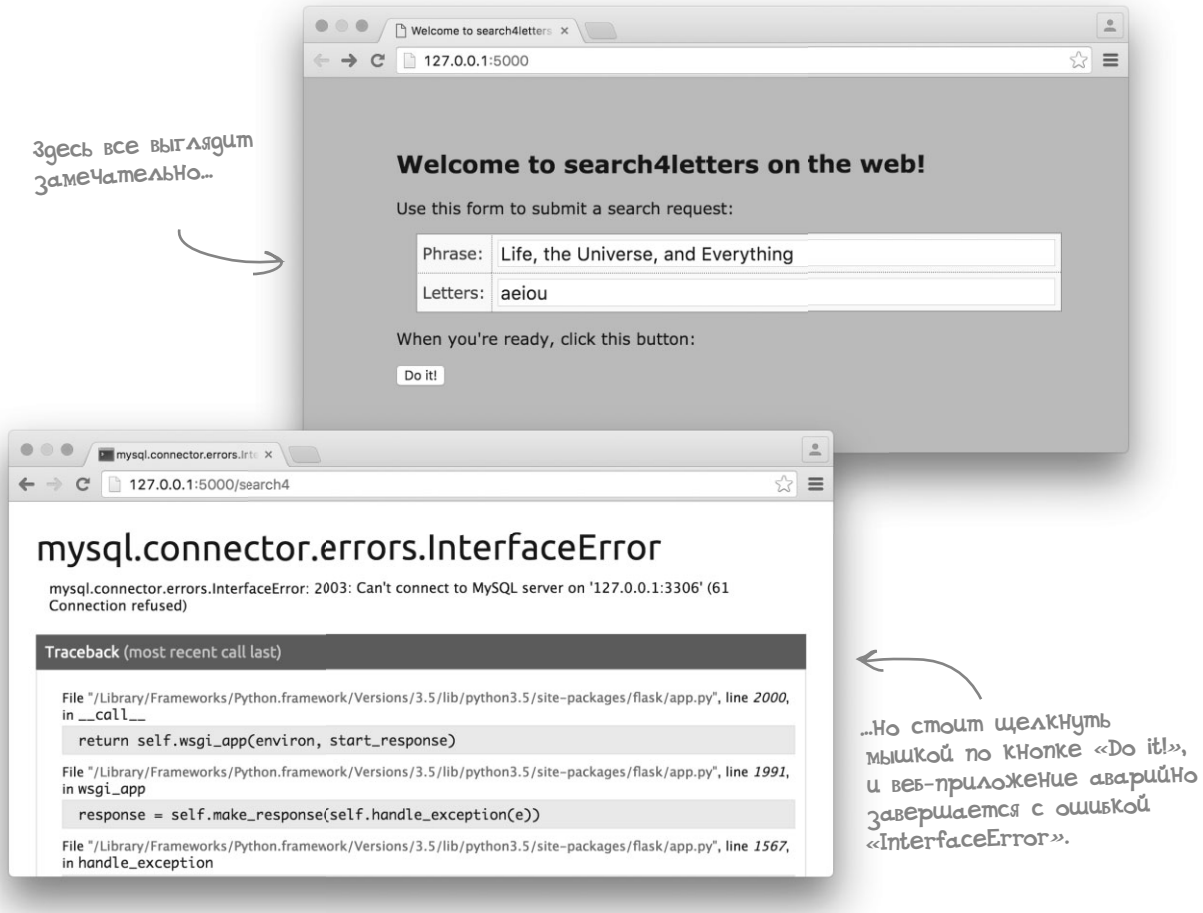
4. Что случится, если этот вызов потерпит неудачу?

Базы данных не всегда доступны

Мы только что выявили четыре потенциальных проблемы в приложении `vsearch4web.py` и допускаем, что их может быть намного больше, но прямо сейчас мы займемся решением только этих четырех проблем. Рассмотрим каждую из них подробнее (здесь и на следующих нескольких страницах мы просто опишем проблемы; *работать над их решением мы будем далее в этой главе*). Прежде всего займемся базой данных:

1. **Что случится, если попытка подключения к базе данных потерпит неудачу?**
   Наше веб-приложение безмятежно предполагает, что база данных всегда находится в работоспособном состоянии и доступна, но это (по ряду причин) может быть не так. В данный момент непонятно, что случится, когда база данных окажется недоступной, так как наш код не предусматривает такой возможности.

Что же случится, если временно *выключить* базу данных? Как показано ниже, веб-приложение благополучно загружается, но стоит только начать что-нибудь делать, как незамедлительно появляется сообщение об ошибке.

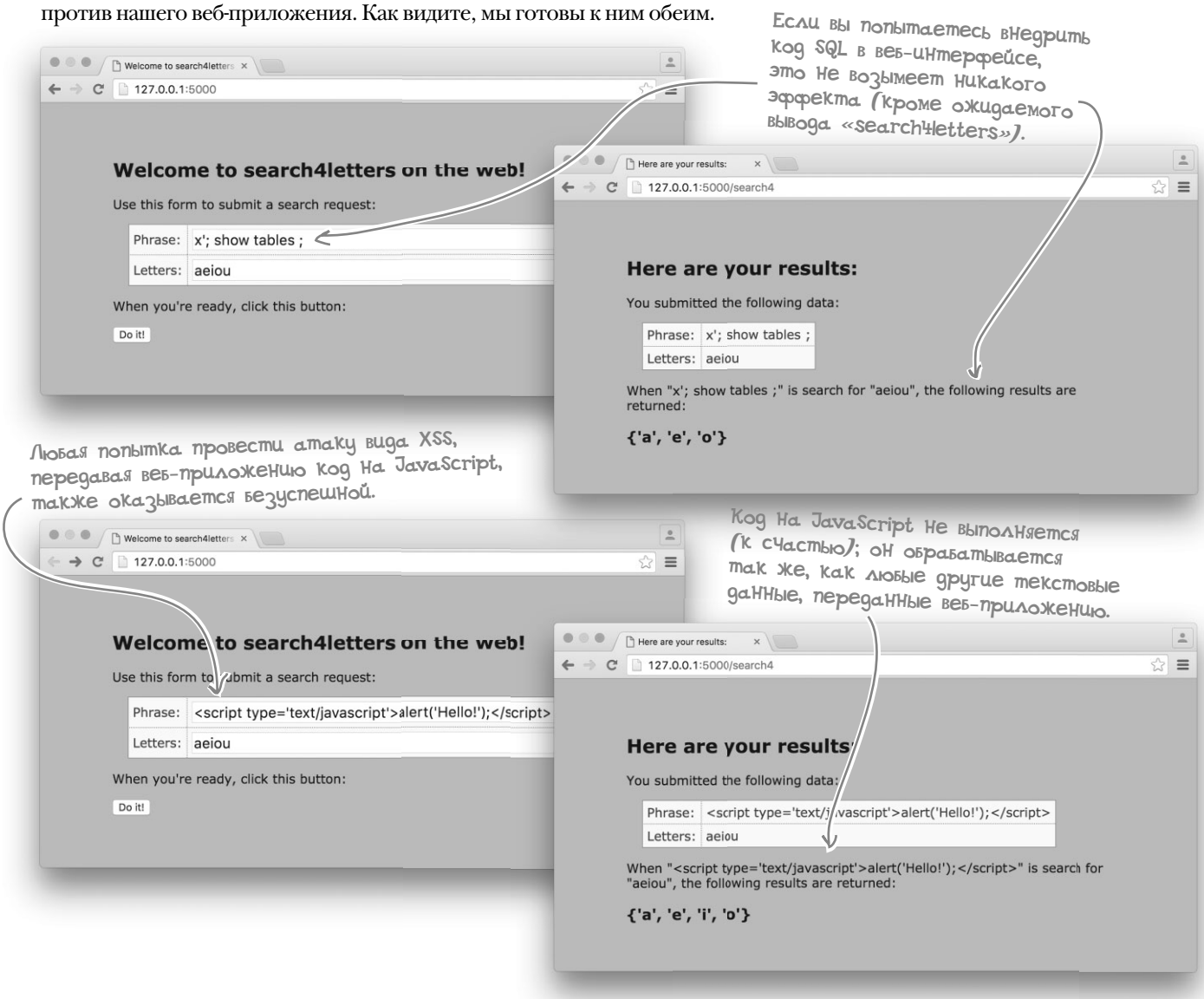


Веб-атаки — настоящая боль

Кроме проблем с базой данных, также нужно побеспокоиться о неприятных индивидуумах, пытающихся доставить неприятности вашему веб-приложению, что ведет нас ко второй проблеме:

2 **Защищено ли наше веб-приложение от веб-атак?**
Фразы *инъекция кода SQL (SQL injection, SQLi)* и *межсайтовый скриптинг (XSS)* обозначают проблемы, которые должны внушать страх каждому веб-разработчику. Первая позволяет атакующим использовать вашу базу данных, в то время как вторая дает им возможность использовать ваш веб-сайт. Есть и другие виды веб-атак, о которых стоит подумать, но эти считаются «большой двойкой».

Посмотрим, что произойдет, если кто-то попробует провести такие атаки против нашего веб-приложения. Как видите, мы готовы к ним обеим.



Ввод-вывод бывает медленным (иногда)

В данный момент наше веб-приложение обменивается информацией с базой данных почти мгновенно, и пользователи замечают лишь небольшую задержку или вообще ее не замечают, пока веб-приложение взаимодействует с базой данных. Но представьте, что взаимодействия с базой данных занимают некоторое время, возможно секунды.

Что произойдет, если выполнение этих SQL-запросов потребует много времени?
База данных может находиться на другой машине, в другом здании, на другом континенте... что случится тогда?

Взаимодействия с базой данных могут потребовать некоторого времени. Фактически, всякий раз когда ваш код взаимодействует с чем-нибудь вовне (например, файлом, базой данных, сетью или чем-то еще), взаимодействие может продолжаться неопределенно долго, и вы не можете повлиять на его продолжительность. В любом случае вы должны понимать, что некоторые операции могут быть продолжительными.

Чтобы продемонстрировать эту проблему, добавим *искусственную* задержку в веб-приложение (с помощью функции `sleep`, которая является частью модуля `time` из стандартной библиотеки). Добавьте эту строку в начало веб-приложения (рядом с другими инструкциями `import`):

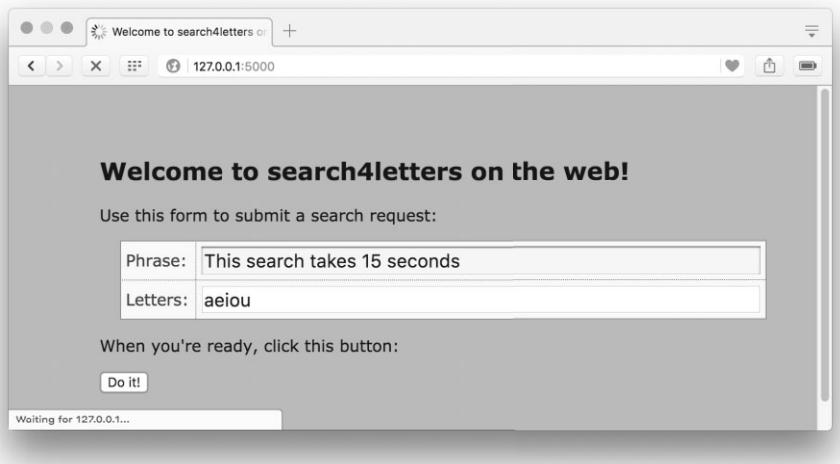
```
from time import sleep
```

Вставив инструкцию `import`, добавьте в функцию `log_request` эту строку кода перед инструкцией `with`:

```
sleep(15)
```

Если теперь перезапустить веб-приложение и инициировать поиск, появится очень заметная задержка, в то время как веб-браузер будет ожидать, пока веб-приложение его догонит. Задержка в 15 секунд покажется вечностью, и большинство пользователей вашего веб-приложения решат, что что-то *сломалось*:

После щелчка мышью на кнопке «Do it!» веб-браузер будет ждать... и ждать... и ждать... и ждать...

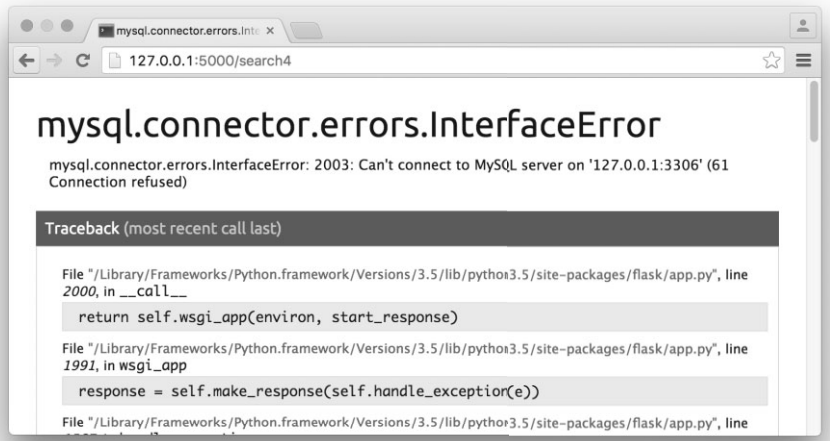


Вызовы функций могут оканчиваться неудачей

Последняя проблема, обозначенная в упражнении в начале главы, относится к вызову функции `log_request` внутри функции `do_search`.

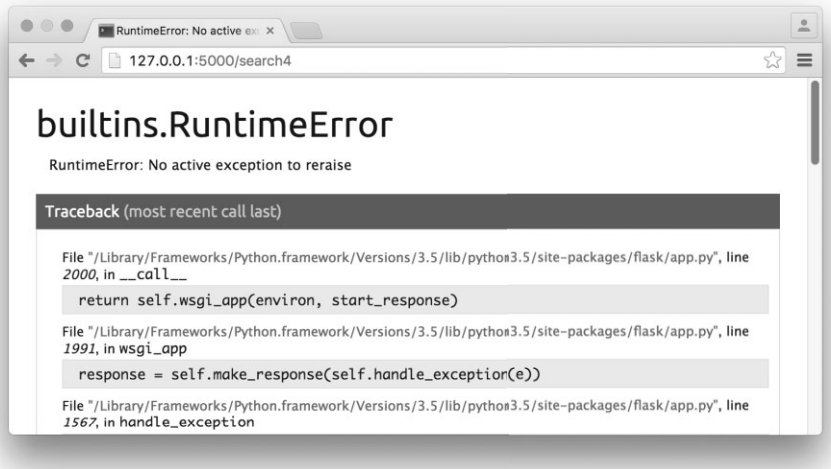
4 **Что случится, если вызов функции потерпит неудачу?**
Никогда нельзя гарантировать, что вызов функции будет успешным, особенно если функция взаимодействует с чем-то вовне.

Мы уже видели, что может случиться, если база данных окажется недоступной — веб-приложение завершится аварийно, с ошибкой `InterfaceError`:



Недоступность базы данных вызывает аварийное завершение веб-приложения.

Также могут проявиться другие проблемы. Чтобы сымитировать другую ошибку, найдите строку **`sleep`**`(15)`, добавленную при обсуждении проблемы 3, и замените ее инструкцией: **`raise`**. Встретив эту инструкцию, интерпретатор сгенерирует ошибку времени выполнения. Если после этого попробовать обратиться к веб-приложению, произойдет *другая* ошибка:



Опять что-то пошло неправильно, веб-приложение снова завершилось аварией.

Прежде чем перелистнуть страницу, удалите инструкцию «raise» из вашего кода, чтобы восстановить нормальную работу веб-приложения.

Исследование выявленных проблем

Мы выявили в приложении `vsearch4web.py` четыре проблемы. Давайте повторно обратимся к каждой и посмотрим, что можно предпринять для их предотвращения.

1. Попытка подключения к базе данных потерпела неудачу

Ошибки случаются всякий раз, когда внешняя система, на которую полагается ваш код, оказывается недоступной. Когда это случается, интерпретатор оповещает об этом ошибкой `InterfaceError`. Можно обнаружить, а затем отреагировать на ошибки этого типа, используя встроенный в Python механизм обработки исключений. Если у вас есть возможность обнаружить ошибку, вы сможете обработать ее.

2. Приложение стало целью атаки

Противодействие атакам против приложений обычно является заботой веб-разработчиков, тем не менее всегда нужно использовать (и обдумывать) приемы, улучшающие устойчивость кода, который вы пишете. В `vsearch4web.py` дела с «большой двойкой» веб-атак, *инъекция SQL* (*SQLi*) и *межсайтовый скриптинг* (*XSS*), кажется, идут хорошо. Это больше счастливая случайность, чем заслуга вашего исходного замысла, так как библиотека Jinja2 по умолчанию предусматривает защиту против *XSS*, экранируя любые потенциально проблематичные строки (вспомните, что код на JavaScript, с помощью которого мы попытались обмануть веб-приложение, не возымел никакого эффекта). Что касается атаки *SQLi*, то использование поддержки параметризованных SQL-запросов в DB-API гарантировало — опять же благодаря тому что модули так спроектированы — защищенность кода от этого класса атак.

Для умников

Если вы хотите узнать больше об атаках SQLi и XSS, Википедия — отличная отправная точка. Посмотрите страницы: *https://https://ru.wikipedia.org/wiki/Внедрение_SQL-кода* и *https://ru.wikipedia.org/wiki/Межсайтовый_скриптинг*. И помните, что есть другие виды атак, которые могут стать источником проблем для приложения; просто эти два — наиболее значительные.

3. Код требует много времени для выполнения

Если код требует много времени для выполнения, вы должны оценить задержку с точки зрения своего пользовательского опыта. Если пользователю задержка не заметна, у вас, вероятно, все в порядке. Однако если пользователь ждет слишком долго, вы должны как-то решить проблему (в противном случае пользователь поймет, что ожидание того не стоит, и может уйти).

4. Вызов функции потерпел неудачу

Генерировать исключения в интерпретаторе могут не только внешние системы, но и ваш код. И когда это случится, вы должны быть готовы обнаружить исключение, а затем восстановить работоспособность, если это необходимо. Для этого используется тот же механизм, что был подсказан в обсуждении проблемы 1 выше.

Итак... с чего *начать*, чтобы решить четыре проблемы? Поскольку для решения проблем 1 и 4 можно использовать один и тот же механизм, именно с него мы и начнем.

Всегда используйте try для кода с высоким риском ошибок

Когда что-то идет не так, Python возбуждает **исключение** времени выполнения. Считайте исключения контролируемым программой сбоем, который обрабатывается интерпретатором.

Как вы видели при исследовании проблем 1 и 4, исключения могут возникать при многих различных обстоятельствах. Фактически, интерпретатор поддерживает массу встроенных типов исключений, и `RuntimeError` (из проблемы 4) является лишь одним примером. Помимо встроенных типов исключений, есть возможность определять собственные, пользовательские исключения, и вы также видели пример: исключение `InterfaceError` (из проблемы 1) определяется модулем драйвера *MySQL Connector*.

Полный список встроенных исключений можно найти по адресу: https://docs.python.org/3/library/exceptions.html

Обнаружить (и по возможности исправить) исключение времени выполнения позволяет встроенная в Python инструкция `try`, способная управлять исключениями, возникшими во время выполнения.

Здесь не происходит ничего примечательного: код открывает указанный файл, извлекает из него данные и выводит их на экран.

```
try_examples.py - /Users/paul/Desktop/_NewBo...

with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)

Ln: 5  Col: 0
```

Чтобы увидеть `try` в действии, рассмотрим фрагмент кода, который может потерпеть неудачу в процессе выполнения. Вот три строки, на первый взгляд безопасные, но потенциально проблематичные.

В этих строках кода нет ничего неправильного — они будут выполняться. Однако этот код потерпит неудачу, если не сможет получить доступ к `myfile.txt`. Файл, например, может отсутствовать или ваш код не имеет необходимых прав доступа на чтение файла. Когда код терпит неудачу, он возбуждает исключение.

Когда возникает ошибка времени выполнения, Python выводит «трассировку» стека вызовов, которая объясняет, что пошло неправильно и где это случилось. В примере интерпретатор думает, что проблема в строке 2.

Упс!

```
Python 3.5.1 Shell

>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py =========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>

Ln: 119  Col: 4
```

Несмотря на неприглядный вид, сообщение с трассировкой стека на самом деле весьма полезно.

Давайте посмотрим, как инструкция `try` может защитить фрагмент кода выше от исключения `FileNotFoundError`.

Перехвата ошибок недостаточно

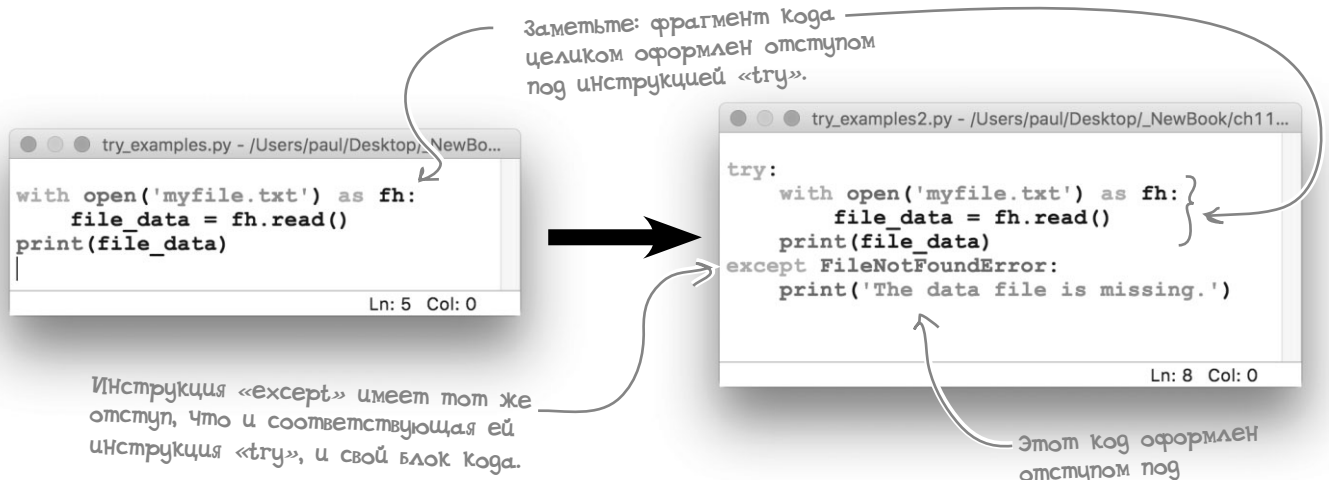
Когда возникает ошибка времени выполнения, **возбуждается** исключение. Если *игнорировать* исключение, которое в этом случае будет называться **неперехваченным**, интерпретатор завершит код и отобразит сообщение об ошибке времени выполнения (как показано в конце предыдущей страницы). С другой стороны, исключения могут **перехватываться** (то есть обрабатываться) инструкцией `try`. Но имейте в виду, что недостаточно перехватывать ошибки времени выполнения, необходимо *также* решить, что делать дальше.

Когда возникает ошибка времени выполнения, ее можно перехватить или не перехватить: «try» позволяет перехватить ошибку, а «except» — сделать что-нибудь с ней.

Возможно, вы решите умышленно игнорировать исключение и продолжите выполнение... скрестив пальцы на удачу. Или, может быть, попытаетесь выполнить некоторый другой код вместо потерпевшего неудачу, и продолжите. Возможно, лучшее, что можно сделать, это записать ошибку в журнал, а затем завершить приложение как можно правильнее. Что бы вы ни решили, вам поможет инструкция `try`.

В наиболее общей форме инструкция `try` позволяет реагировать на любое исключение, возникшее в ходе выполнения кода. Чтобы защитить код с помощью `try`, поместите его внутрь блока `try`. Если код в блоке `try` возбудит исключение, его выполнение прервется и запустится код в соответствующем блоке `except`. Блок `except` — это то место, где определяется, что должно произойти потом.

Давайте изменим код с предыдущей страницы, чтобы он отображал короткое сообщение всякий раз, когда возникает исключение `FileNotFoundError`. Слева показан прежний код, а справа — исправленный так, чтобы использовать преимущество, которое предлагают `try` и `except`:



Обратите внимание: раньше было три строки кода, а сейчас стало шесть, что может выглядеть расточительным, но на самом деле это не так. Оригинальный фрагмент кода все еще существует как сущность; он превратился в блок инструкции `try`. Инструкция `except` и ее блок кода — это новый код. Итак, посмотрим, какое различие внесли эти исправления.

Пробная поездка

Давайте запустим версию кода с `try... except`. Если файл `myfile.txt` существует и доступен вашему коду для чтения, его содержимое появится на экране. Если нет, интерпретатор возбудит исключение времени выполнения. Мы уже знаем, что `myfile.txt` не существует, но сейчас, вместо вывода неприглядного сообщения с трассировкой стека, как в прошлый раз, интерпретатор запустит код обработки исключения и на экране появится более дружелюбное сообщение (хотя наш код *все еще* терпит неудачу):

Когда мы в первый раз запустили фрагмент кода, интерпретатор сгенерировал это уродливое сообщение.

```
Python 3.5.1 Shell
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py =========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>>
                                                                    Ln: 17  Col: 4
```

Новая версия выводит более дружелюбное сообщение, благодаря «try» и «except».

Исключения могут быть разными...

Новое поведение лучше, но что случится, если `myfile.txt` существует, а код не имеет права на чтение из него? Чтобы увидеть это, создадим файл, а затем установим права доступа к нему так, чтобы сымитировать эту ситуацию. На этот раз новый код вывел следующее.

Вот черт! Снова это уродливое сообщение с трассировкой, так как возникло исключение «PermissionError».

```
Python 3.5.1 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
                                                                    Ln: 24  Col: 4
```

Одна инструкция try, но несколько except

Чтобы защититься от другого возникшего исключения, надо просто добавить в инструкцию `try` еще один блок `except`, указав обрабатываемое исключение и добавив в него какой-нибудь код, который вы сочтете необходимым. Вот другая, дополненная версия кода, которая обрабатывает исключение `PermissionError` (если оно возникнет).

Кроме исключения «FileNotFoundError», этот код обрабатывает также «PermissionError».

try_examples3.py - /Users/paul/Desktop/_NewBook/ch11/try_...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
```

Ln: 10 Col: 0

Код в блоках «except» может выполнять нужные операции. Но сейчас они просто выводят дружественные сообщения.

В ходе выполнения этот исправленный код все еще возбуждает исключение `PermissionError`. Но теперь вместо уродливого сообщения выводится более дружелюбное.

Python 3.5.1 Shell

```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
This is not allowed.
>>>
```

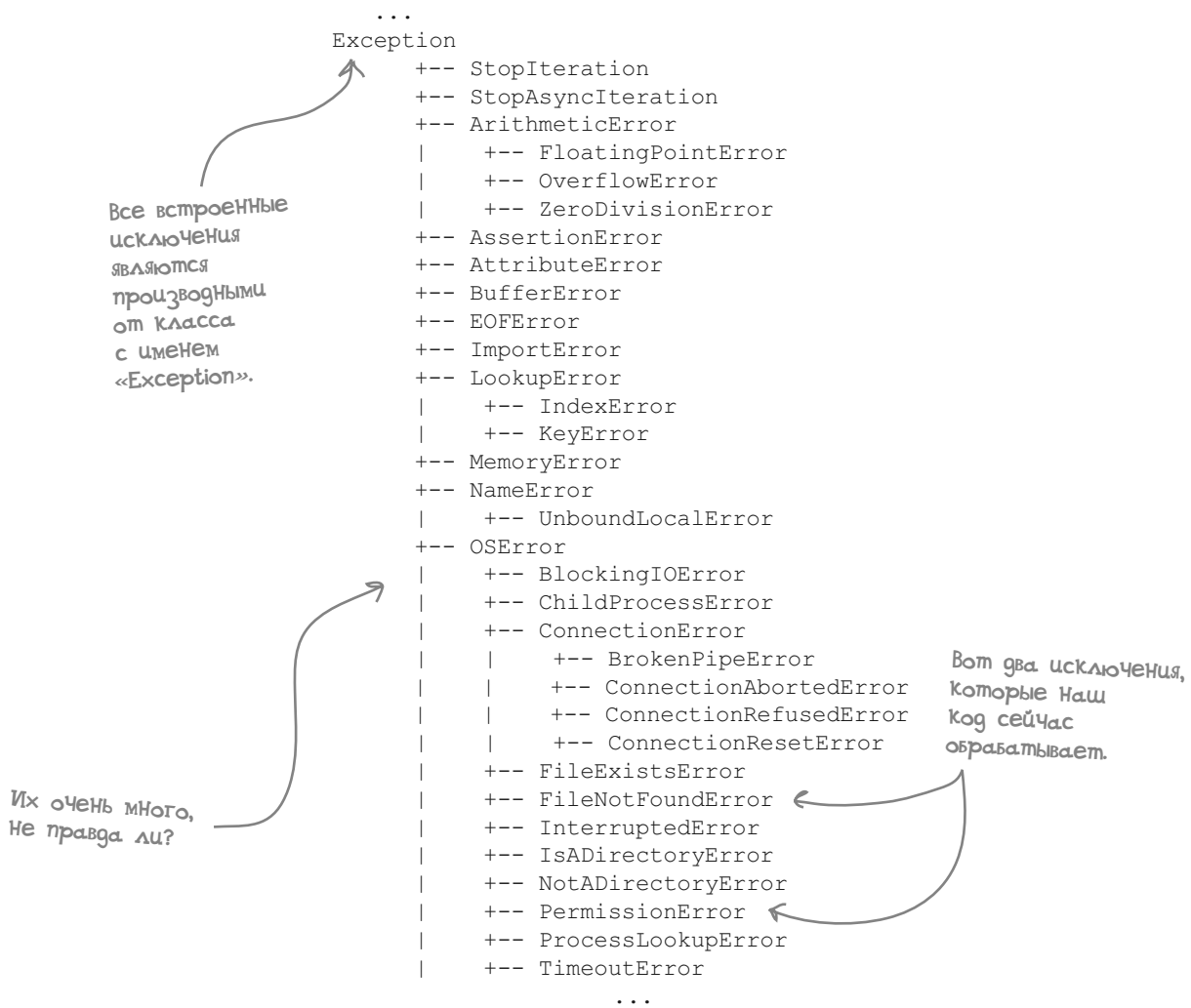
Ln: 27 Col: 4

Так намного лучше.

Все здорово: теперь вы сможете управлять происходящим, когда файл, с которым вы собираемся работать, окажется в другом месте (или вообще не существует) или будет недоступен (из-за нехватки прав доступа). Но что случится, если возникнет исключение, которого вы не ожидаете?

Многое может пойти не так

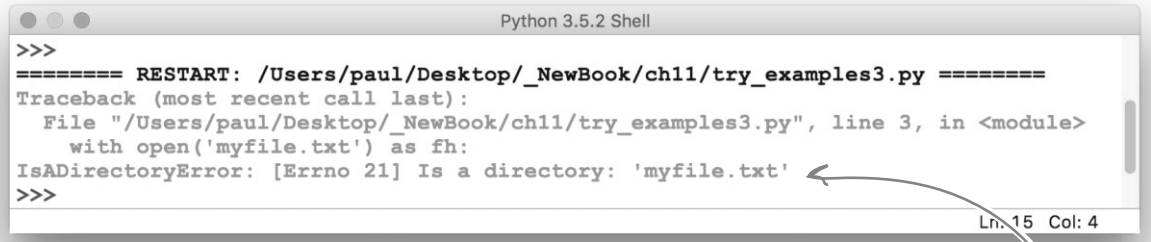
Прежде чем дать ответ на вопрос, сформулированный в конце предыдущей страницы, — *что случится, если возникнет исключение, которого вы не ожидаете?* — давайте посмотрим на некоторые исключения, встроенные в Python 3 (следующий фрагмент скопирован прямо из документации по Python). Не удивляйтесь их многообразию.



Было бы безумием пытаться писать отдельный блок `except` для каждого из этих исключений времени выполнения, так как некоторые из них могут никогда не возникнуть. С другой стороны, некоторые *могут* случиться, поэтому о них придется позаботиться. Вместо того чтобы пытаться обработать каждое исключение *индивидуально*, Python позволяет определить блок `except` **на все случаи жизни**, который будет запускаться всякий раз, когда возникнет исключение времени выполнения, не указанное явно.

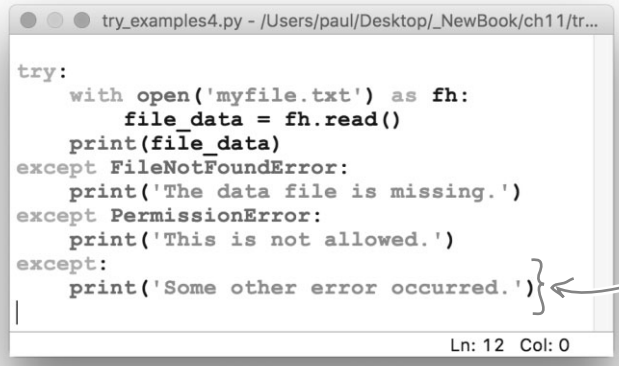
Обработчик любых исключений

Посмотрим, что произойдет, когда возникнет какая-нибудь другая ошибка. Чтобы сымитировать именно такой случай, мы превратили файл `myfile.txt` в папку. Давайте узнаем, что произойдет, если мы запустим код.



Возникло другое исключение.

Возникло другое исключение. Можно, конечно, добавить еще один блок `except`, который будет запускаться в ответ на исключение `IsADirectoryError`, но давайте пойдем другим путем и определим обработчик любых ошибок времени выполнения. Он будет срабатывать, когда возникнет *любое* другое исключение (отличное от тех двух, которые мы уже обрабатываем). Для этого добавьте в конец еще одну инструкцию `except`.

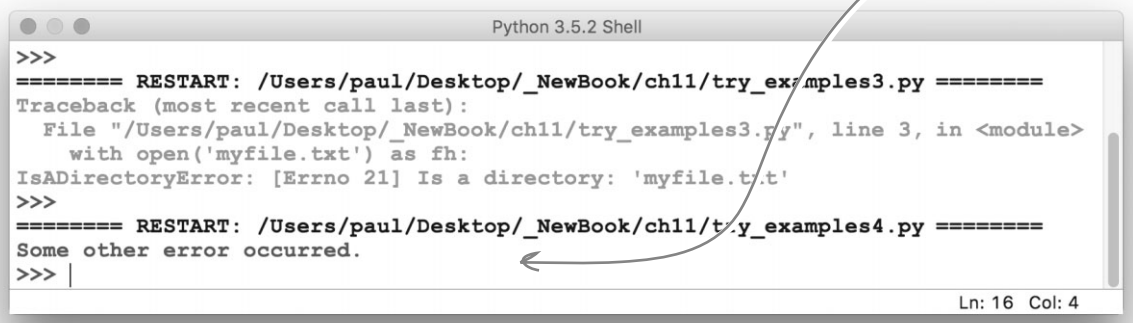


Инструкция «except» является «универсальной»: она не ссылается на конкретное исключение.

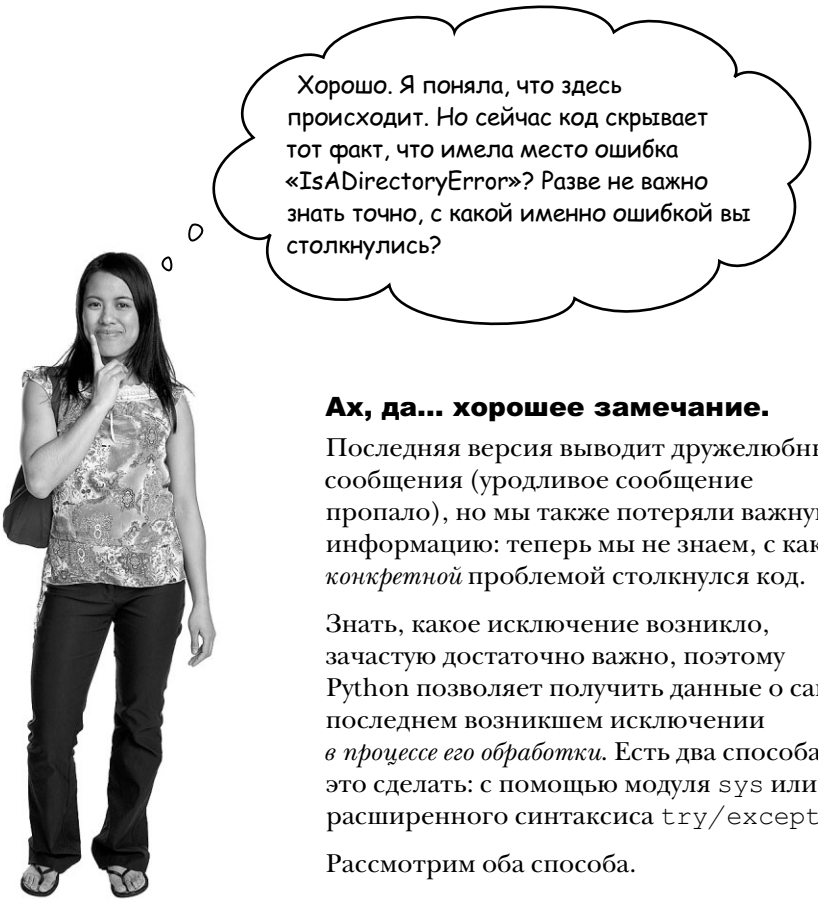
Обработчик любых исключений.

Вместо уродливого сообщения интерпретатора эта исправленная версия выводит более дружественное. Не важно, какое именно другое исключение возникнет, этот код обработает его, благодаря дополнительному универсальному блоку `except`.

Это выглядит лучше.



Мы ничего не потеряли?



Ах, да... хорошее замечание.

Последняя версия выводит дружелюбные сообщения (уродливое сообщение пропало), но мы также потеряли важную информацию: теперь мы не знаем, с какой *конкретной* проблемой столкнулся код.

Знать, какое исключение возникло, зачастую достаточно важно, поэтому Python позволяет получить данные о самом последнем возникшем исключении *в процессе его обработки*. Есть два способа это сделать: с помощью модуля `sys` или расширенного синтаксиса `try/except`.

Рассмотрим оба способа.

Это не Глупые вопросы

В: Можно ли создать универсальный обработчик исключений, который ничего не делает?

О: Да. Бывает заманчиво добавить такой блок `except` в конец инструкции `try`:

```
except:
    pass
```

Пожалуйста, старайтесь не делать так. Такой блок `except` *игнорирует* любые другие исключения (необоснованно надеясь, что если что-нибудь не замечать, то оно уладится само собой). Это опасная практика, так как — по крайней мере — в результате неожиданного исключения должно появиться сообщение на экране. Так что убедитесь, что всегда пишете код с проверкой ошибок, который обрабатывает исключения, а не игнорирует их.

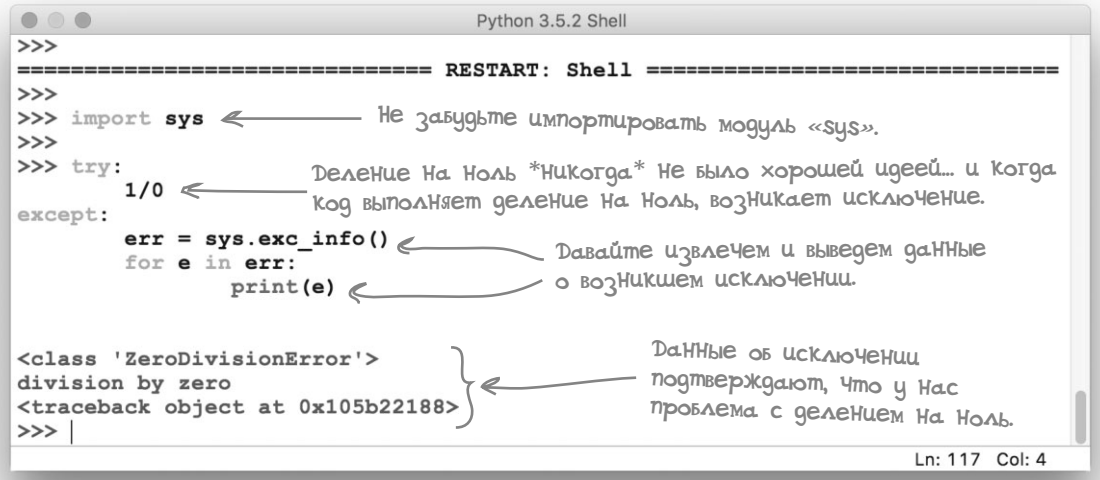
Узнаем об исключениях из «sys»

Стандартная библиотека включает модуль с именем `sys`, который открывает доступ к *внутренним механизмам* интерпретатора (набору переменных и функций, доступных во время выполнения).

Одна из таких функций — `exc_info` — возвращает информацию об исключении, которое в данный момент обрабатывается. Вызов `exc_info` возвращает кортеж с тремя значениями, первое из которых содержит **тип** исключения, второе — **значение** исключения, и третье — **объект трассировки**, открывающий доступ к сообщению трассировки (если он вам нужен). Когда текущее исключение недоступно, `exc_info` возвращает кортеж с тремя пустыми значениями, который выглядит вот так: `(None, None, None)`.

Чтобы узнать больше о «sys», посмотрите https://docs.python.org/3/library/sys.html

Теперь, зная все это, поэкспериментируем в оболочке >>>. В следующем сеансе IDLE мы написали код, который всегда терпит неудачу (деление на ноль *никогда* не было хорошей идеей). Универсальный блок `except` использует функцию `sys.exc_info`, чтобы извлечь и отобразить данные, относящиеся к активному в данный момент исключению.



Есть заглянуть в объект трассировки, чтобы узнать о случившемся больше, то чувствуется, что тут слишком много работы. Все, что мы действительно хотим знать, это *тип* возникшего исключения.

Чтобы сделать это (и вашу жизнь) проще, Python расширяет синтаксис `try/except` и упрощает получение информации, возвращаемой функцией `sys.exc_info`. При этом не требуется помнить о необходимости импортировать модуль `sys` или разбираться с кортежем, возвращаемым этой функцией.

Как упоминалось несколькими страницами выше, интерпретатор организует исключения в иерархию, где все классы исключений наследуют единственный общий класс с именем `Exception`. Давайте воспользуемся преимуществом этой иерархической организации и перепишем наш универсальный обработчик исключений.

Вспомним иерархию исключений, представленную ранее.

```
...
Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
          ...
```

Универсальный обработчик, повторение

Рассмотрим наш текущий код, который недвусмысленно идентифицирует два типа обрабатываемых исключений (`FileNotFoundError` и `PermissionError`), а также включает универсальный блок `except` (для обработки всех остальных исключений).

Код работает, но не сообщает никакой дополнительной информации, если возникнет какое-то неожиданное исключение.

try_examples4.py - /Users/paul/Desktop/_NewBook/ch11/tr...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')
```

Ln: 12 Col: 0

Обратите внимание: при обработке *конкретное* исключение идентифицируется своим именем, следующим за ключевым словом `except`. Наравне с конкретным исключением, можно идентифицировать *классы* исключений, используя любые имена из иерархии.

Например, если достаточно знать, что произошла арифметическая ошибка (в отличие от конкретной ошибки деления на ноль), можно определить блок `except ArithmeticError`, который будет перехватывать исключения `FloatingPointError`, `OverflowError` и `ZeroDivisionError`. По аналогии, если определить блок `except Exception`, он будет перехватывать *любые* ошибки.

Вспомним, что все эти исключения являются производными от класса «Exception».

```
...
Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
          ...
```

Но как это может помочь... ведь мы и так перехватываем все ошибки с «пустой» инструкцией `except`? Это правда: мы это сделали. Но инструкцию `except Exception` можно дополнить ключевым словом `as`, позволяющим присвоить текущий объект исключения переменной (`err` — очень популярное имя в такой ситуации), и создать более информативное сообщение об ошибке. Рассмотрим другую версию кода, которая использует `except Exception as`.

try_examples5.py - /Users/paul/Desktop/_NewBook/ch11/try_examples5.py...

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except Exception as err:
    print('Some other error occurred:', str(err))
```

Ln: 12 Col: 0

В отличие от «пустой» универсальной инструкции «except», показанной выше, эта позволяет присвоить объект исключения переменной «err».

Значение «err» затем используется как часть дружественного сообщения (поскольку сообщать обо всех исключениях — всегда хорошая идея).

Пробная поездка

Применив последние изменения к инструкции `try/except`, убедимся, что она все еще работает как ожидается, а затем вернемся к `vsearch4web.py` и применим в веб-приложении все, что мы узнали об исключениях.

Сначала убедимся, что код выводит правильное сообщение, если файл не существует.

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
                                                                    Ln: 7  Col: 4
```

«myfile.txt» отсутствует.

Если файл есть, но вы не имеете прав доступа к нему, будет возбуждено другое исключение.

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
                                                                    Ln: 10  Col: 4
```

Файл существует, но вы не можете читать его.

Любое другое исключение обрабатывается универсальным обработчиком, который выводит дружественное сообщение.

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
                                                                    Ln: 23  Col: 4
```

Возникло некоторое другое исключение. Теперь нетрудно понять, что файл фактически оказался папкой.

Наконец, если все закончилось хорошо, блок `try` выполнится без ошибок, и на экране появится содержимое файла.

```
Python 3.5.2 Shell
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
This is not allowed.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ========
Empty (well... except for this line).

>>>
                                                                    Ln: 27  Col: 4
```

Ура! Исключения не возникли, поэтому блок «try» выполнился до конца.

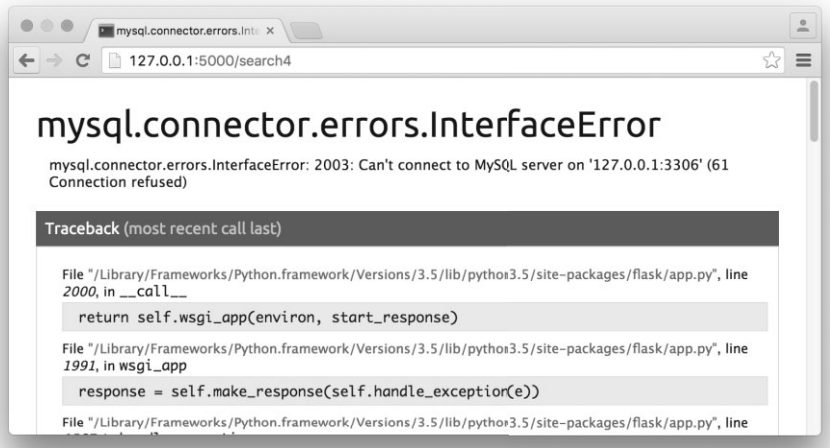
Назад к нашему веб-приложению

Итак, в начале главы мы выявили проблему с вызовом `log_request` внутри функции `do_search` в `vsearch4web.py`. Если конкретно, нас взволновал вопрос: что делать, если вызов `log_request` завершится неудачей.

```
        ...
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
        ...
```

4. Что случится, если этот вызов потерпит неудачу?

В ходе исследований мы узнали, что этот вызов может завершаться неудачей, если база данных окажется недоступной или произойдет какая-нибудь другая ошибка. В случае ошибки (любого типа) веб-приложение отвечает недружественной страницей с сообщением, которое, по всей вероятности, будет смущать (а не просвещать) пользователей.



Не хотелось бы, чтобы пользователи веб-приложения видели это сообщение.

Журналирование каждого веб-запроса важно для нас, но это не совсем то, что действительно интересует пользователей веб-приложения; все они хотят видеть результаты поиска. Поэтому давайте изменим код веб-приложения так, чтобы любые исключения, возникшие внутри `log_request`, обрабатывались *молча*.

Тихая обработка исключений

Нет: «молчать» не значит «игнорировать».

Когда мы предложили обрабатывать исключения *молча*, мы имели в виду обрабатывать незаметно для пользователей веб-приложения. В данный момент пользователи *замечают*, что веб-приложение завершается аварийно и показывает сбивающую с толку и — будем честны — *пугающую* страницу с сообщением об ошибке.

Пользователи не должны беспокоиться об ошибке, возникшей в `log_request`, но вы заставляете их это делать. Итак, давайте изменим код, чтобы исключения, возникшие в `log_request`, оставались незаметными для пользователей (то есть обрабатывались тихо), но *были* заметными для вас.

Это не Глупые вопросы

В: **Не станет ли мой код труднее для чтения и понимания из-за всех этих** `try/except`**?**

О: Действительно, пример, первоначально включавший три простые и понятные строки кода на Python, мы дополнили семью строками, которые — на первый взгляд — не делают ничего сверх того, что делают первые три строки. Однако они необходимы для защиты кода от возможных исключений, а инструкция `try/except`, в общем случае, считается лучшим способом сделать это. Со временем ваш мозг научится выделять все самое важное (код, который действительно выполняет полезную работу) в блоках `try` и отфильтровывать блоки `except`, необходимые для обработки исключений. Если хотите понять код, использующий `try/except`, всегда сначала читайте блок `try`, чтобы узнать, что код делает, а затем смотрите на блоки `except`, чтобы выяснить, что случится, когда что-то пойдет не так.

Заточите карандаш

Давайте добавим в функцию `do_search` инструкцию `try/except`, заключив в нее вызов `log_request`. Чтобы не усложнять, определим единственный, универсальный обработчик исключений, который будет выводить полезное сообщение в стандартный вывод (с помощью `print`). В обработчике исключений можно переопределить стандартную реакцию веб-приложения на исключения — вывод недружественной страницы с сообщением об ошибке.

Вот как сейчас выглядит код `log_request`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Эту строку нужно защитить от неудачи (ошибки времени выполнения).

В пропуски ниже вставьте код, реализующий обработку любых исключений, которые могут возникнуть в вызове `log_request`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))

    .....

        .....

    .....

        .....

    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Не забудьте вызвать «log_request» в коде, который вы добавите.

Заточите карандаш
Решение

План был такой: добавить в функцию `do_search` инструкцию `try/except`, заключив в нее вызов `log_request`. Чтобы не усложнять, мы решили определить единственный, универсальный обработчик исключений, который будет выводить полезное сообщение в стандартный вывод (с помощью `print`).

Вот как сейчас выглядит код `log_request`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

В пропуски ниже мы вставили код, реализующий обработку любых исключений, которые могут возникнуть в вызове `log_request`.

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Вызов «log_request» перемещен в блок новой инструкции «try».

Универсальный обработчик.

Когда возникнет ошибка времени выполнения, обработчик выведет это сообщение на экран для администратора. Пользователь ничего не увидит.

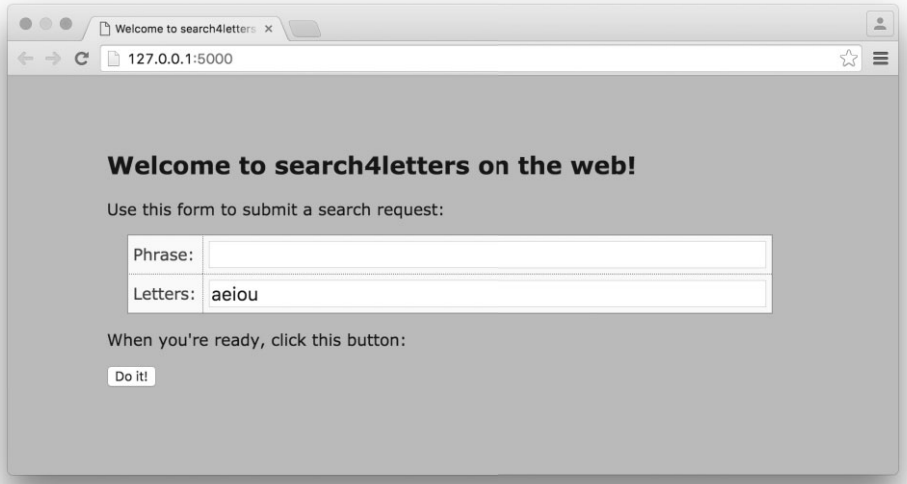
— (Расширенная) Пробная поездка, 1 из 3 —

Давайте добавим код универсального обработчика исключений в `vsearch4web.py` и совершим расширенную поездку по нашему веб-приложению (на следующих нескольких страницах), чтобы увидеть, какие отличия внес новый код. Раньше, когда что-нибудь шло не так, пользователя приветствовала недружественная страница с сообщением об ошибке. Сейчас ошибка обрабатывается «молча». Если вы еще этого не сделали, запустите `vsearch4web.py`, а затем используйте любой браузер, чтобы зайти на домашнюю страницу веб-приложения.

```
$ python3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with fsevents reloader
 * Debugger is active!
 * Debugger pin code: 184-855-980
```

Веб-приложение запущено, работает и ожидает известий от браузера...

Откройте домашнюю страницу веб-приложения.



В терминале, где запущен код, вы должны увидеть примерно следующее.

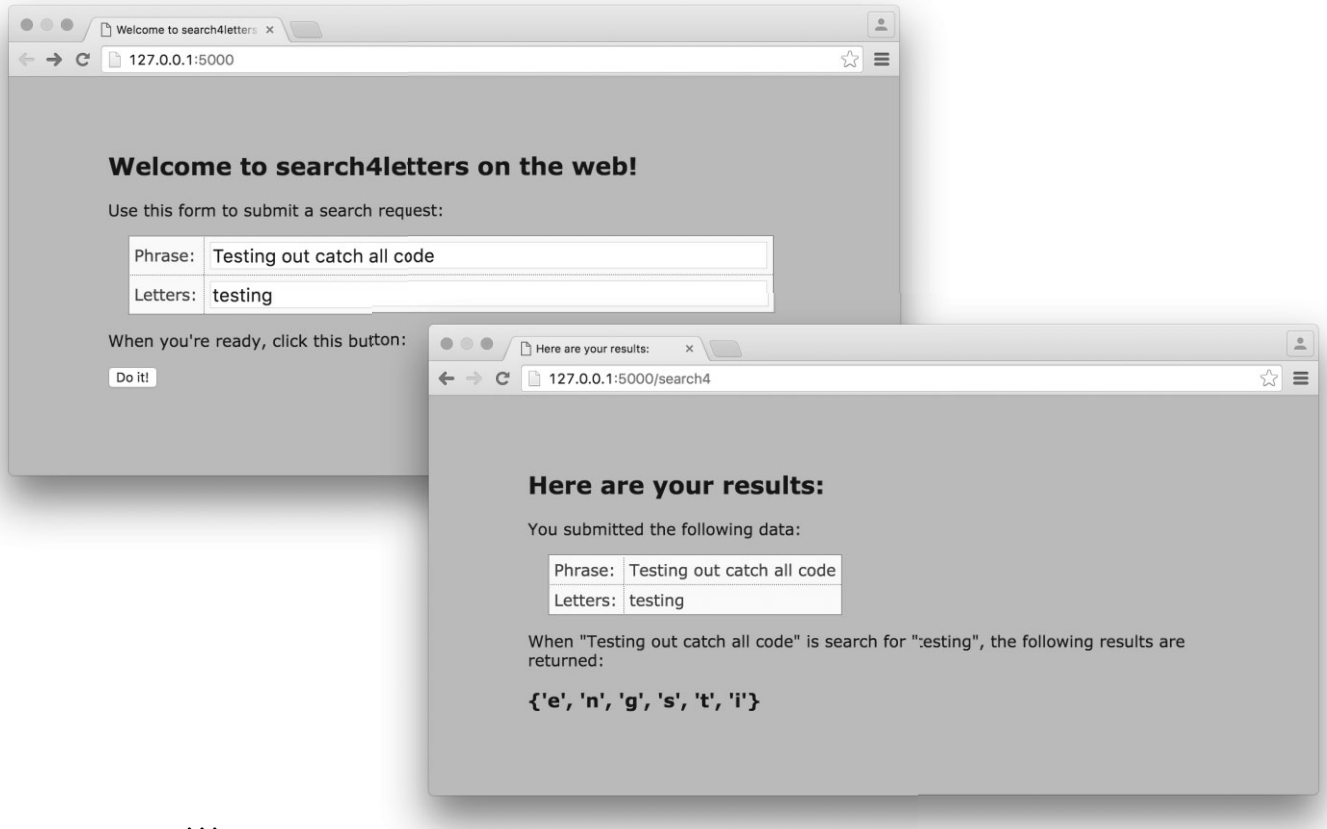
```
    ...
 * Debugger pin code: 184-855-980
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -
```

Эти числа 200 подтверждают, что веб-приложение запущено и работает (и обслуживает свою домашнюю страницу). Пока все хорошо.

Кстати, не беспокойтесь о числе 404... мы не определили файл «favicon.ico» для веб-приложения (эта строка сообщает, что файл не находится, когда браузер запрашивает его).

— (Расширенная) Пробная поездка, 2 из 3 —

Выключим нашу базу данных, чтобы сымитировать ошибку. В результате всякий раз, когда веб-приложение попытается взаимодействовать с базой данных, будет происходить ошибка. Так как наш код молча перехватывает все ошибки, сгенерированные функцией `log_request`, пользователь веб-приложения не будет знать, что журналирование не выполняется. Универсальный обработчик выводит на экран сообщения с описанием проблемы. Как и ожидалось, после ввода фразы и щелчка на кнопке «Do it!» веб-приложение показывает в браузере результаты поиска, а в окне терминала — «тихое» сообщение об ошибке. Обратите внимание, что несмотря на ошибку времени выполнения, веб-приложение продолжает работать и благополучно обрабатывает запросы к URL */search.*




```
...
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -
***** Logging failed with this error: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [14/Jul/2016 10:55:55] "POST /search4 HTTP/1.1" 200 -
```

Сообщение, сгенерированное универсальным обработчиком исключений. Пользователь веб-приложения не увидит его.

Теперь ошибка не приводит к аварийному завершению веб-приложения. Другими словами, поиск работает (но пользователь веб-приложения не знает, что операция журналирования окончилась неудачей).

— (Расширенная) Пробная поездка, 3 из 3 —

Не имеет значения, какая ошибка возникнет в вызове `log_request`, — универсальный обработчик ее обработает.

Мы перезапустили базу данных, а затем попытались подключиться с неверным именем пользователя. Вы можете вызвать эту ошибку, изменив словарь `dbconfig` в `vsearch4web.py`, присвоив `vsearchwrong` ключу `user`.

```
            ...
app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearchwrong',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }
            ....
```

Перезапустив веб-приложение и выполнив поиск, вы увидите подобное сообщение в терминале.

```
  ...
***** Logging failed with this error: 1045 (28000): Access denied for user 'vsearchwrong'@
'localhost' (using password: YES)
```

Давайте изменим значение `user` обратно на `vsearch` и попытаемся получить доступ к несуществующей таблице, изменив имя таблицы в SQL-запросе, используемом в функции `log_request`, на `logwrong` (вместо `log`).

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into logwrong
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        ...
```

Перезапустив веб-приложение и выполнив поиск, вы увидите в терминале подобное сообщение:

```
  ...
***** Logging failed with this error: 1146 (42S02): Table 'vsearchlogdb.logwrong' doesn't exist
```

Теперь изменим имя таблицы обратно на `log` и, как заключительный пример, добавим в функцию `log_request` (прямо перед выражением `with`) инструкцию `raise`, которая генерирует пользовательское исключение.

```
def log_request(req: 'flask_request', res: str) -> None:
    raise Exception("Something awful just happened.")
    with UseDatabase(app.config['dbconfig']) as cursor:
        ...
```

Перезапустив веб-приложение в последний раз и выполнив поиск, вы увидите в терминале следующее сообщение.

```
  ...
***** Logging failed with this error: Something awful just happened.
```

Обработка других ошибок в базе данных

Функция `log_request` использует диспетчер контекста `UseDatabase` (из модуля `DBcm`). Сейчас, защитив вызов `log_request`, можете быть уверенными, что любая проблема, связанная с ошибками в базе данных, будет перехвачена (и обработана) универсальным обработчиком исключений.

Однако функция `log_request` — не единственное место, где веб-приложение взаимодействует с базой данных. Функция `view_the_log` извлекает записи из журнала в базе данных, чтобы отобразить их на экране.

Вспомним, как выглядит функция `view_the_log`.

Весь этот код тоже нуждается в защите.

```
        ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(_SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
        ...
```

Этот код также может потерпеть неудачу, так как он взаимодействует с базой данных. Однако, в отличие от `log_request`, функция `view_the_log` не вызывается из кода `vsearch4web.py`; ее вызывает Flask от нашего имени. Этот означает, что мы не можем написать код, защищающий вызов `view_the_log`, так как эту функцию вызывает фреймворк Flask, а не мы.

Если нет возможности защитить вызов `view_the_log`, остается одно — защитить вызов диспетчера контекста `UseDatabase`. Прежде чем перейти к реализации такой защиты, давайте подумаем, что может пойти не так.

- Сервер базы данных может быть недоступен.
- Вы можете не иметь возможности записывать информацию в работающую базу данных.
- После успешной регистрации запрос к базе данных может потерпеть неудачу.
- Что-нибудь иное (неожиданное) может случиться.

Этот список проблем точно такой же, как в случае с `log_request`.

«Больше ошибок» значит «больше except»?

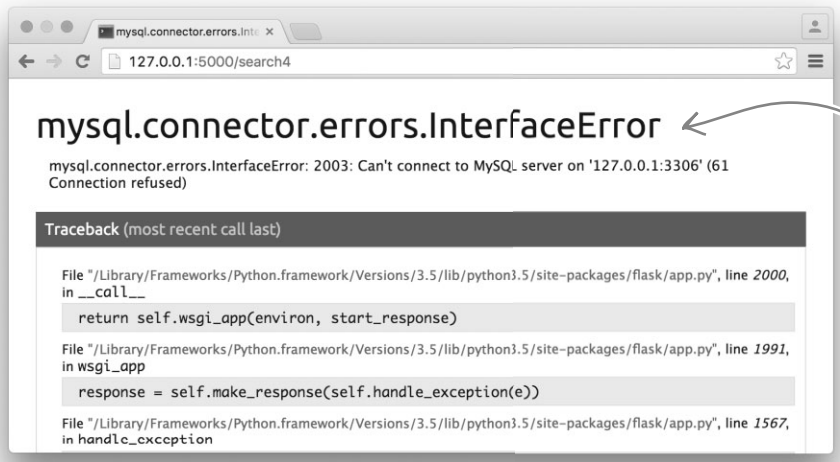
Зная все то, что мы сейчас знаем о `try/except`, мы можем добавить код в функцию `view_the_log`, чтобы защитить обращение к диспетчеру контекста `UseDatabase`.

```
        ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            ...
    except Exception as err:
        print('Something went wrong:', str(err))
```

Другой универсальный обработчик исключений

Остальной код функции.

Стратегия применения универсального обработчика, безусловно, работает (в конце концов, именно ее мы использовали в `log_request`). Однако все может усложниться, если вы решите сделать что-нибудь отличное от универсальной обработки исключений. Например, решите иначе реагировать на конкретные ошибки базы данных, такие как «База данных не найдена». Как говорилось в начале главы, в этом случае MySQL генерирует исключение `InterfaceError`.



Это исключение возбуждается, когда код не может найти базу данных.

Вы можете добавить инструкцию `except`, обрабатывающую исключение `InterfaceError`, но для этого необходимо импортировать модуль `mysql.connector`, который определяет данное конкретное исключение.

На первый взгляд это не кажется сложным. Но является.

Избегайте тесных связей в коде

Предположим, что вы решили добавить инструкцию except, которая будет защищать от ошибки недоступности базы данных. Вы можете изменить код view_the_log, чтобы он выглядел примерно так:

```
        ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            ...

    except mysql.connector.errors.InterfaceError as err:
        print('Is your database switched on? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
            ...
```

Остальной код функции.

Дополнительная инструкция «except» для обработки конкретного исключения.

Если вы не забудете добавить import mysql.connector в начало файла, эта дополнительная инструкция except будет работать. Когда драйвер не сможет отыскать базу данных, дополнительный код позволит веб-приложению напомнить вам о необходимости проверить, включена ли база данных.

Новый код работает, и вы видите, что здесь происходит... что тут может не нравиться?

Проблема в том, что код в vsearch4web.py сейчас очень **тесно связан** с базой данных *MySQL* и непосредственно использует модуль *MySQL Connector*. До того как была добавлена эта вторая инструкция except, код vsearch4web.py взаимодействовал с базой данных через модуль DBcm (разработанный ранее в книге). В частности, диспетчер контекста UseDatabase реализует удобную **абстракцию**, отделяющую код vsearch4web.py от базы данных. Если в какой-то момент в будущем потребуется заменить *MySQL* на *PostgreSQL*, достаточно будет изменить только модуль DBcm, а *не* весь код, использующий UseDatabase. Однако, создавая код, подобный тому, что показан выше, вы тесно связываете (то есть привязываете) веб-приложение с базой данных MySQL при помощи инструкции import mysql.connector. Кроме того, ваша новая инструкция except ссылается на mysql.connector.errors.InterfaceError.

Если понадобится код, тесно связанный с базой данных, его лучше поместить внутрь модуля DBcm. В этом случае веб-приложение сможет использовать более универсальный интерфейс модуля DBcm, вместо конкретного, ориентированного на определенную базу данных (и ограниченного им).

Давайте посмотрим, что даст нашему веб-приложению перемещение кода except в DBcm.

Модуль DBcm, еще раз

Последний раз мы рассматривали модуль `DBcm` в главе 9 (мы создали его, чтобы использовать инструкцию `with` для работы с базой данных *MySQL*). Тогда мы уклонились от любых обсуждений обработки ошибок (удобнее было игнорировать проблему). Сейчас, после знакомства с функцией `sys.exc_info`, вы должны лучше понимать, что означают аргументы метода `__exit__` в `UseDatabase`.

Это код диспетчера контекста в «DBcm.py».

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Сейчас, после знакомства с функцией «exc_info», вам должно быть понятно, что эти аргументы метода ссылаются на информацию об исключении.

Как вы помните, `UseDatabase` реализует три метода:

- `__init__` позволяет выполнить настройки *перед тем*, как `with` начнет выполнение;
- `__enter__` выполняется, когда выражение `with` *запустится*;
- `__exit__` гарантированно выполняется, когда блок кода `with` *завершается*.

По меньшей мере, таково ожидаемое поведение, когда все идет по плану. Когда что-то идет не так, это поведение **меняется**.

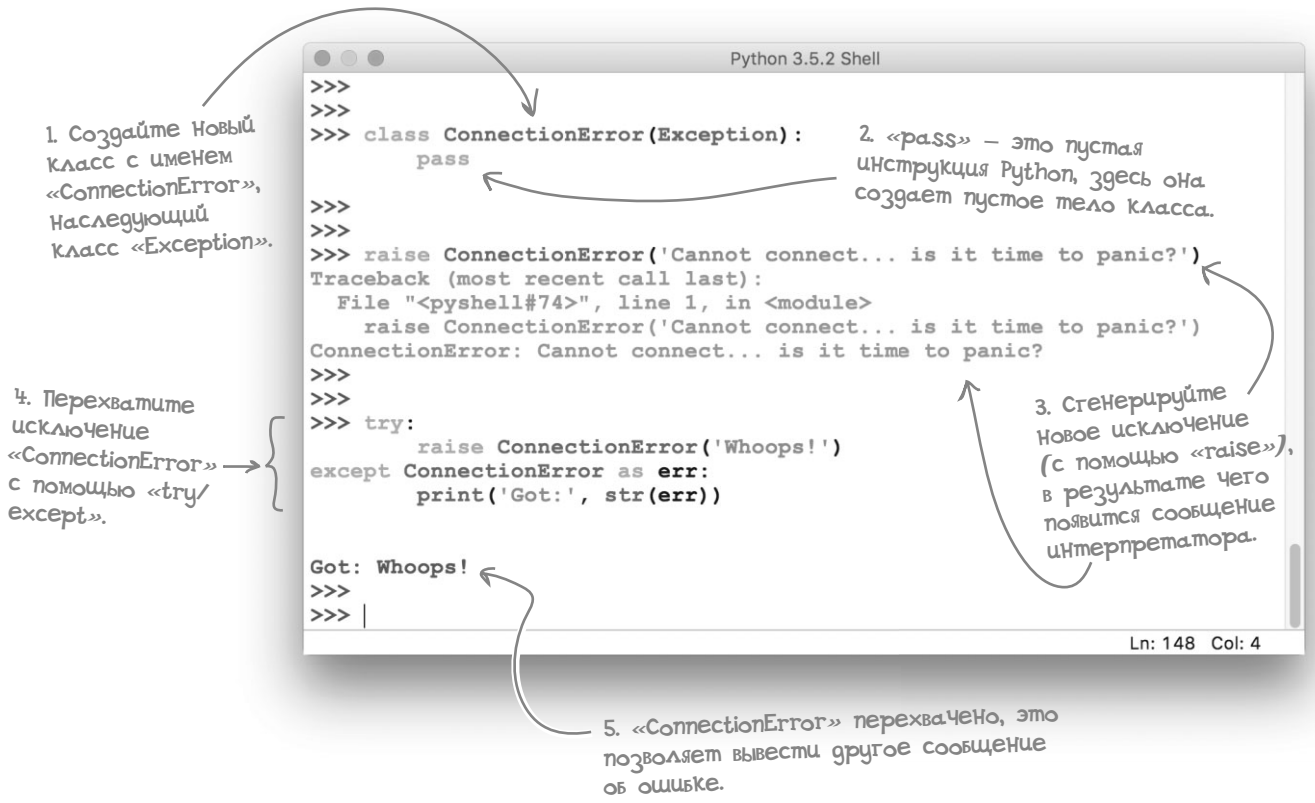
Например, если исключение возникло в методе `__enter__`, выражение `with` завершится, и любая последующая обработка в `__exit__` *отменится*. В этом есть определенный смысл: если `__enter__` столкнулся с проблемой, `__exit__` не может предполагать, что контекст выполнения правильно инициализирован и настроен (поэтому отказ от вызова `__exit__` выглядит вполне разумным).

Самая большая проблема, которая может возникнуть в методе `__enter__`, — недоступность базы данных. Потратим немного времени на изменение `__enter__` и обработаем ситуацию, сгенерировав собственное исключение, если попытка соединиться с базой данных потерпела неудачу. Как мы уже делали однажды, изменим `view_the_log` и добавим проверку нестандартного исключения вместо конкретного `mysql.connector.errors.InterfaceError`.

Создание собственного исключения

Создать собственное исключение — что может быть проще? Выберите подходящее имя, а затем определите пустой класс, наследующий встроенный в Python класс `Exception`. После этого собственное исключение можно возбудить с помощью ключевого слова `raise`, а затем перехватить (и обработать) с помощью `try/except`.

Короткий сеанс в >>> IDLE демонстрирует, как действуют пользовательские исключения. В этом примере мы создали исключение с именем `ConnectionError`, возбудили его (с помощью `raise`), а затем перехватили с помощью `try/except`. Читайте аннотации в порядке нумерации и (следуя за ними в этом порядке) введите код, который мы ввели в командную строку >>>.



Пустой класс на самом деле не совсем пустой...

Описывая класс `ConnectionError` как «пустой», мы немножко отклонились от истины. Инструкция `pass` гарантирует лишь отсутствие *нового* кода, ассоциированного с классом `ConnectionError`, но фактически `ConnectionError` **наследует** встроенный класс `Exception`, то есть все атрибуты и методы `Exception` доступны в `ConnectionError` (что делает его не совсем пустым). Это объясняет, почему `ConnectionError` работает с `raise` и `try/except` именно так, как ожидается.

Заточите карандаш

1 Давайте изменим модуль `DBcm` так, чтобы пользовательское исключение `ConnectionError` возникало при каждой неудачной попытке соединиться с базой данных. Вот текущий код `DBcm.py`. В пропуски впишите код, генерирующий исключение `ConnectionError`.

Определите свое исключение.

```
import mysql.connector

.....

    .....

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':

        .....
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor

        .....

            .....
    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Добавьте код, генерирующий «ConnectionError».

2 Изменив код модуля `DBcm`, опишите все изменения, которые следует внести в `vsearch4web.py`, чтобы воспользоваться преимуществами нового исключения `ConnectionError`.

Опишите, какие изменения следует внести в этот код сейчас, когда существует исключение «ConnectionError».

```
from DBcm import UseDatabase
import mysql.connector
        ...
                                the_row_titles=titles,
                                the_data=contents,)
    except mysql.connector.errors.InterfaceError as err:
        print('Is your database switched on? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'
```

Заточите карандаш
Решение

1. Вы изменили модуль `DBcm`, добавив возбуждение собственного исключения `ConnectionError` при каждой неудачной попытке соединиться с базой данных. Вы изменили `DBcm.py`, добавив код, генерирующий исключение `ConnectionError`.

Определение своего исключения в виде «пустого» класса, наследующего «Exception».

```
import mysql.connector

class ConnectionError(Exception):
    pass

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Внутри «DBcm.py» ссылаемся на исключения для конкретной базы данных по их полным именам.

Новая конструкция «try/except» защищает код подключения к базе данных.

Возбуждение собственного исключения.

2. После изменения кода в модуле `DBcm` нужно было описать все изменения, которые следует внести в `vsearch4web.py`, чтобы воспользоваться преимуществами нового исключения `ConnectionError`.

Больше не нужно импортировать «mysql.connector» (так как «DBcm» делает это за вас).

```
from DBcm import UseDatabase, ConnectionError
~~import mysql.connector~~

...

                the_row_titles=titles,
                the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'
```

Не забудьте импортировать исключение «ConnectionError» из «DBcm».

Измените первую инструкцию «except», чтобы она обрабатывала «ConnectionError», а не «InterfaceError».

Пробная поездка

Посмотрим, чем отличается новый код. Итак, мы переместили код обработки исключения, характерного для MySQL, из `vsearch4web.py` в `DBcm.py` (и заменили его кодом, который обрабатывает наше собственное исключение `ConnectionError`). Изменилось ли поведение программы?

Вот сообщения, которые генерировала предыдущая версия `vsearch4web.py`, когда не могла найти базу данных.

```
        ...
Is your database switched on? Error: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:21:51] "GET /viewlog HTTP/1.1" 200 -
```

А вот сообщения, которые генерирует последняя версия `vsearch4web.py`, когда не может найти базу данных:

```
        ...
Is your database switched on? Error: 2003: Can't connect to MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:22:58] "GET /viewlog HTTP/1.1" 200 -
```

Да. На первый взгляд все то же самое.

Однако несмотря на то, что вывод текущей версии `vsearch4web.py` ничем не отличается от предыдущей, за кулисами все выглядит *совсем по-другому*.

Если вы решите поменять базу данных *MySQL* на *PostgreSQL*, вам не придется беспокоиться об изменении какого-либо кода в `vsearch4web.py`, так как весь код, зависящий от базы данных, находится в `DBcm.py`. Пока изменения в `DBcm.py` будут поддерживать тот же *интерфейс*, что и предыдущие версии модуля, вы можете менять базы данных SQL, когда вам захочется. Возможно, сейчас это не кажется важным, но если объем кода в `vsearch4web.py` вырастет до сотен, тысяч или десятков тысяч строк, это будет иметь значение.

Что еще может пойти не так с «DBcm»?

Даже если ваша база данных работает, все равно что-то может пойти не так.

Например, учетные данные, использованные для доступа к базе данных, могут быть некорректными. Если это так, метод `__enter__` снова потерпит неудачу, на этот раз с исключением `mysql.connector.errors.ProgrammingError`.

Исключение может возникнуть в реализации диспетчера контекста `UseDatabase`, поскольку никогда нельзя гарантировать, что он будет выполняться корректно. Исключение `mysql.connector.errors.ProgrammingError` также будет генерироваться, если запрос к базе данных (код SQL, который вы выполняете) содержит ошибку.

Сообщение об ошибке, связанное с ошибкой в SQL-запросе, отличается от сообщения, связанного с ошибкой в учетных данных, но исключение то же самое: `mysql.connector.errors.ProgrammingError`. В отличие от ошибок с учетными данными, ошибки в SQL-запросе порождают исключения во время выполнения инструкции `with`. Этот означает, что защищаться от этого исключения придется в нескольких местах. Возникает вопрос: где?

Чтобы ответить на этот вопрос, еще раз посмотрим на код `DBcm`:

```
import mysql.connector

class ConnectionError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Этот код может возбудить исключение «ProgrammingError».

Но как быть с исключениями, возникающими в блоке кода «with»? Они возникают *после* того, как метод «__enter__» завершится, но *прежде* чем запустится метод «__exit__».

Вы можете поддаться искушению обрабатывать все исключения, возникшие в блоке кода `with` с помощью инструкций `try/except` *внутри* `with`, но такая стратегия откинет вас назад к написанию тесно связанного кода. Но посмотрите вот на что: когда исключение возбуждается в блоке `with` и *не* перехватывается, инструкция `with` передает информацию о неперехваченном исключении в метод `__exit__` диспетчера контекста, где можно как-то на это отреагировать.

Добавляем новые пользовательские исключения

Давайте добавим в `DBcm.py` еще два нестандартных исключения.

Первому дадим имя `CredentialsError` и будем возбуждать в ответ на появление исключения `ProgrammingError` внутри метода `__enter__`. Второе назовем `SQLError` и будем возбуждать в ответ на исключение `ProgrammingError` в методе `__exit__`.

Определить новые исключения легко: надо только добавить два новых, пустых класса исключений в начало `DBcm.py`.

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:
    def __init__(self, configuration: dict):
        self.config = configuration
              ...
```

Два дополнительных класса определяют два новых исключения.

`CredentialsError` может возникать в ходе выполнения `__enter__`, поэтому добавим в него код, отражающий это обстоятельство. Напомним, что ошибка в имени пользователя MySQL или в пароле приведет к появлению `ProgrammingError`.

Добавьте этот код в метод «__enter__» для обработки любых проблем с регистрацией.

```
              ...
        try:
            self.conn = mysql.connector.connect(**self.config)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)
        except mysql.connector.errors.ProgrammingError as err:
            raise CredentialsError(err)

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Новый код в `DBcm.py` возбуждает исключение `CredentialsError` при попытке передать неправильное имя пользователя или пароль базе данных (MySQL). Изменить код в `vsearch4web.py` — ваша следующая задача.

Учетные данные для вашей базы данных правильные?

Учтем последние изменения в `DBcm.py` и отредактируем код `vsearch4web.py`, уделяя особенное внимание функции `view_the_log`. Но сперва добавим `CredentialsError` в список импорта из `DBcm`, в начале `vsearch4web.py`.

Не забудьте импортировать новое исключение.

```
...
from DBcm import UseDatabase, ConnectionError, CredentialsError
...
```

Исправив строку `import`, добавьте новый блок кода `except` в функцию `view_the_log`. Это ничуть не сложнее, чем добавить поддержку `ConnectionError`.

```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                    from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'
```

Добавьте этот код в «view_the_log», чтобы отследить момент, когда ваш код использует неправильные имя пользователя или пароль для MySQL.

Здесь нет ничего нового — мы просто повторяем то, что уже делали, когда добавляли `ConnectionError`. Без сомнения, если сейчас попытаться подключиться к базе данных с неверным именем пользователя (или паролем), веб-приложение выведет соответствующее сообщение, подобное этому.

```
...
User-id/Password issues. Error: 1045 (28000): Access denied for user 'vsearcherror'@'localhost'
(using password: YES)
127.0.0.1 - - [25/Jul/2016 16:29:37] "GET /viewlog HTTP/1.1" 200 -
```

Сейчас, когда ваш код знает все о «CredentialsError», можно вывести для этого исключения более конкретное сообщение об ошибке.

Обработка SQLError отличается

Оба исключения — `ConnectionError` и `CredentialsError` — возбуждаются внутри метода `__enter__`. Когда появляется одно из этих исключений, блок кода в инструкции `with` **не** выполняется.

Если все хорошо, блок кода `with` выполнится как обычно.

Напомним, что инструкция `with` в функции `log_request` использует диспетчер контекста `UseDatabase` (из модуля `DBcm`) для записи в базу данных.

Нам нужно беспокоиться о том, что случится, если с этим кодом что-нибудь пойдет не так (то есть с кодом в блоке «with»).

```
with UseDatabase(app.config['dbconfig']) as cursor:
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Если (по какой-то причине) SQL-запрос содержит ошибку, модуль *MySQL Connector* сгенерирует исключение `ProgrammingError`, подобное тому, что возбуждалось в ходе выполнения метода `__enter__` диспетчера контекста. Однако поскольку это исключение возникнет *внутри* диспетчера контекста (то есть внутри инструкции `with`) и *не* будет перехвачено, оно будет передано назад в метод `__exit__` в виде трех аргументов: *типа* исключения, *значения* исключения и *трассировки* стека, связанной с исключением.

Взгляните на имеющийся код `__exit__` в `DBcm`, и вы увидите, что эти три аргумента готовы и ожидают использования.

Три аргумента с информацией об исключении готовы к использованию.

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Если исключение возникло в блоке `with` и не было перехвачено, диспетчер контекста завершит выполнение блока `with` и перепрыгнет в метод `__exit__`. Зная это, можно написать код, проверяющий исключение, интересующее ваше приложение. Однако если исключение не возникло, все три аргумента (`exc_type`, `exc_value` и `exc_traceback`) будут содержать значение `None`. В противном случае они будут заполнены деталями о возникшем исключении.

«None» — это аналог null в Python.

Давайте используем эту особенность, чтобы сгенерировать `SQLError`, когда в блоке кода `with` с диспетчером контекста `UseDatabase` что-то пойдет не так.

Будьте осторожны с позиционированием кода

Чтобы определить тип неперехваченного исключения, возникшего в блоке инструкции `with`, проверьте аргумент `exc_type` в методе `__exit__` и внимательно выберите место для нового кода.

Впервые слышу. Какая разница, куда я помещу код проверки «exc_type»? Разве она есть?

Да, разница есть.

Чтобы понять почему, рассмотрим метод `__exit__` диспетчера контекста, куда можно поместить код, который **обязательно** будет выполнен *после* выполнения блока `with`. В конце концов, это поведение определяется протоколом управления контекстом.

Такое поведение сохраняется, даже когда в блоке `with` вашего диспетчера контекста возникает исключение. Это значит, что если вы планируете добавить код в метод `__exit__`, то его лучше поместить *после* существующего кода. Таким образом вы гарантируете, что существующий код будет выполнен (и сохраните семантику протокола управления контекстом).

Давайте взглянем еще раз на код в методе `__exit__`, но уже с учетом его. Наша задача — добавить некоторый код, который будет возбуждать исключение `SQLError`, если `exc_type` указывает, что возникло исключение `ProgrammingError`.

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Если добавить код сюда и этот код возбудит исключение, то три существующие строки не будут выполняться.

Если добавить код *после* трех существующих строк, то «__exit__» сделает свои дела *прежде*, чем обработает переданное исключение.

Возбуждаем SQLError

К данному моменту мы уже добавили класс исключения `SQLError` в начало файла `DBcm.py`.

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config
              ...
```

Вот здесь вы добавили исключение «SQLError».

Имея класс исключения `SQLError`, нам остается только добавить код в метод `__exit__`, который проверит интересующий нас тип исключения в `exc_type` и при необходимости возбудит `SQLError`. Это настолько просто, что мы противимся обычному побуждению *Head First* превратить создание необходимого кода в упражнение, так как у нас нет желания оскорблять чей-либо интеллект. Итак, вот код, который вам нужно добавить в метод `__exit__`.

Если возникло «ProgrammingError», возбудить «SQLError».

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
    if exc_type is mysql.connector.errors.ProgrammingError:
        raise SQLError(exc_value)
```

Чтобы получить **дополнительную защиту** и дать возможность обработать любое другое неожиданное исключение, переданное в `__exit__`, можно добавить в конец метода `__exit__` блок кода `elif`, который повторно сгенерирует неожиданное исключение для вызывающего кода.

```
          ...
    self.conn.close()
    if exc_type is mysql.connector.errors.ProgrammingError:
        raise SQLError(exc_value)
    elif exc_type:
        raise exc_type(exc_value)
```

Этот «elif» повторно возбудит любое другое исключение, которое может возникнуть.

Пробная поездка

Пользуясь поддержкой исключения `SQLError` в `DBcm.py`, добавим другой блок кода `except` в функцию `view_the_log`, чтобы перехватить все возникающие исключения `SQLError`.

Добавьте этот код в функцию «view_the_log» внутри веб-приложения «vsearch4web.py».

```
            ...
except ConnectionError as err:
    print('Is your database switched on? Error:', str(err))
except CredentialsError as err:
    print('User-id/Password issues. Error:', str(err))
except SQLError as err:
    print('Is your query correct? Error:', str(err))
except Exception as err:
    print('Something went wrong:', str(err))
return 'Error'
```

Когда вы сохраните `vsearch4web.py`, веб-приложение перезапустится и будет готово для проверки. Если попытаться выполнить SQL-запрос, содержащий ошибки, код выше обработает исключение.

```
    ...
Is your query correct? Error: 1146 (42S02): Table 'vsearchlogdb.logerror' doesn't exist
127.0.0.1 - - [25/Jul/2016 21:38:25] "GET /viewlog HTTP/1.1" 200 -
```

Больше нет обобщенного исключения «ProgrammingError» из MySQL Connector, так как сейчас пользовательский код обработки исключений перехватывает эти ошибки.

Аналогично, если случится что-нибудь неожиданное, универсальный обработчик в веб-приложении возьмет управление и выведет соответствующее сообщение:

```
    ...
Something went wrong: Some unknown exception.
127.0.0.1 - - [25/Jul/2016 21:43:14] "GET /viewlog HTTP/1.1" 200 -
```

Если случится что-нибудь неожиданное, ваш код это обработает.

С кодом обработки исключения, добавленным в веб-приложение, не важно, какая ошибка времени выполнения возникнет. Веб-приложение продолжит функционировать, не пугая пользователей страшной или сбивающей с толку страницей ошибки.

Действительно, хорошо, что код берет обобщенное исключение «ProgrammingError», генерируемое модулем MySQL Connector, и превращает его в два пользовательских исключения, имеющих конкретный смысл для веб-приложения.

Да, это так. И это очень удобно.

Подведение итогов: добавление надежности

Давайте остановимся на минутку, чтобы вспомнить, что мы собирались сделать в этой главе. В попытках сделать код нашего веб-приложения более надежным, мы ответили на четыре вопроса, относящиеся к четырем выявленным проблемам. Давайте еще раз рассмотрим каждый вопрос и отметим, что мы сделали.

1. **Что случится, если попытка подключения к базе данных потерпит неудачу?**
   Мы создали новое исключение с именем `ConnectionError`, которое возбуждается, когда драйверу не удается найти базу данных. Затем мы использовали `try/except` для обработки `ConnectionError`.

2. **Защищено ли наше веб-приложение от веб-атак?**
   Это была «счастливая случайность», но наш выбор *Flask* плюс *Jinja2* вместе со спецификацией DB-API защищает веб-приложение от большинства печально известных веб-атак. Итак, да, веб-приложение защищено от *некоторых* веб-атак (но нет от всех).

3. **Что случится, если что-нибудь потребует много времени?**
   Мы не ответили на этот вопрос, а только показали, что произойдет, когда веб-приложению потребуется 15 секунд, чтобы ответить на запрос пользователя: пользователь будет вынужден ждать (или, что более вероятно, ему надоест, и он уйдет).

4. **Что случится, если вызов функции потерпит неудачу?**
   Мы использовали `try/except`, чтобы защитить вызов функции. Это позволяет контролировать, что увидит пользователь веб-приложения, когда что-нибудь пойдет не так.

Что случится, если что-нибудь потребует много времени?

Этот вопрос возник в ходе исследования вызова `cursor.execute`, который выполнялся в функциях `log_request` и `view_the_log`, когда мы делали первое упражнение в начале главы. Хотя мы уже работали с обеими функциями, отвечая на вопросы 1 и 4, выше, мы с ним еще не закончили.

Обе функции, `log_request` и `view_the_log`, используют диспетчер контекста `UseDatabase` для выполнения SQL-запросов. Функция `log_request` **записывает** результаты поиска в базу данных, а функция `view_the_log` **читает** их из базы данных.

Вопрос: *что делать, если запись или чтение потребует много времени?*

Что ж, как и со многими вещами в мире программирования, есть варианты.

Как быть с ожиданием? Есть варианты...

Решение, как поступить с кодом, который заставляет пользователей ждать — при чтении или при записи — может стать сложным. Поэтому мы приостановим обсуждение и отложим решение до следующей, короткой главы.

Фактически, следующая глава такая короткая, что нет основания давать ей собственный номер (так мы и поступили), но в ней представлен достаточно сложный материал, чтобы обосновать его выделение в отдельную главу: механизм работы `try/except`. Так что давайте немного задержимся, прежде чем обсудить оставшуюся часть проблемы 3: *что случится, если что-нибудь потребует много времени?*

Да. Ирония нам не чужда.

Мы просим вас *подождать* с изучением особенностей обработки «ожидания» в вашем коде.

Но вы и так узнали очень много нового в этой главе, и мы думаем, что важно сделать паузу, чтобы ваш мозг усвоил информацию о `try/except`.

Итак, мы хотим, чтобы вы остановились и сделали короткий перерыв... после того как просмотрите весь код из этой главы.

Код 11-й главы, 1 из 3

Это «try_example.py».

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except Exception as err:
    print('Some other error occurred:', str(err))
```

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)
        except mysql.connector.errors.ProgrammingError as err:
            raise CredentialsError(err)

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
        if exc_type is mysql.connector.errors.ProgrammingError:
            raise SQLError(exc_value)
        elif exc_type:
            raise exc_type(exc_value)
```

Версия «DBcm.py» с поддержкой исключений.

Код 11-й главы, 2 из 3

Эта версия «vsearch4web.py» заставляет пользователей ждать...

```
from flask import Flask, render_template, request, escape, session
from flask import copy_current_request_context

from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
from checker import check_logged_in

from time import sleep

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15)  # This makes log_request really slow...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                    (phrase, letters, ip, browser_string, results)
                    values
                    (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
```

Хорошо бы защитить эту инструкцию «with», как мы защитили инструкцию «with» в «view_the_log» (на следующей странице).

Остальная часть «do_search» в начале следующей страницы.

Код 11-й главы, 3 из 3

```
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results))
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                    from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Some unknown exception.")
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except SQLError as err:
        print('Is your query correct? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'


app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Оставшаяся часть функции «do_search».

11¾ немного о многопоточности

Обработка ожидания

Иногда для выполнения кода может потребоваться много времени.

В одном случае это становится проблемой, в другом — нет. Если код работает «за кулисами» и ему нужно 30 секунд, чтобы сделать какие-то свои дела, ожидание не превращается в проблему. Однако если приложению требуется 30 секунд, чтобы ответить пользователю, это заметят все. Решение проблемы зависит от того, какие действия пытаетесь выполнить вы (и кто именно ожидает). В этой короткой главе мы обсудим некоторые варианты, а затем рассмотрим одно из решений данной проблемы: *как быть, если что-то требует слишком много времени?*

Ожидание: что делать?

Когда вы пишете код, который может заставить ваших пользователей ждать, хорошенько подумайте, что вы пытаетесь сделать. Вот некоторые точки зрения.

Может быть, это тот случай, когда ожидание при записи *отличается* от ожидания при чтении, особенно если учесть, как работает наше веб-приложение?

Посмотрим еще раз на SQL-запросы в `log_request` и `view_the_log`, чтобы увидеть, как они используются.

Как выполняются запросы к базе данных?

В функции `log_request` мы использовали SQL-инструкцию `INSERT`, чтобы сохранить детали веб-запроса в базе данных. Когда вызывается `log_request`, она **ждет**, пока `cursor.execute` выполнит `INSERT`.

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into log
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        cursor.execute(_SQL, (req.form['phrase'],
                              req.form['letters'],
                              req.remote_addr,
                              req.user_agent.browser,
                              res, ))
```

В этой точке работа веб-приложения «блокируется», пока база данных не выполнит свои внутренние дела.

Для умников

Код, ожидающий завершения чего-нибудь внешнего, называется «блокирующим кодом» — выполнение вашей программы **блокируется**, пока ожидание не закончится. Как правило, если блокирующему коду требуется много времени — это плохо.

То же происходит в функции `view_the_log`, которая также **ждет**, пока выполняется SQL-запрос `SELECT`.

```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                      from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
            ...
```

Здесь веб-приложение тоже «блокируется», ожидая ответа от базы данных.

Чтобы сэкономить место, мы не показали весь код «view_the_log». Код обработки исключений все еще находится здесь.

Обе функции — блокирующие. Однако посмотрим внимательнее, что в них происходит *после* вызова `cursor.execute`. Вызов `cursor.execute` в `log_request` — это последнее, что она делает, тогда как `view_the_log` обрабатывает результаты `cursor.execute` в оставшейся части функции.

Давайте рассмотрим последствия этого различия.

Инструкции INSERT и SELECT базы данных отличаются

Если, прочитав заголовок, вы подумали: «Конечно же, они отличаются!», будьте уверены: мы (ближе к концу книги) не сошли с ума.

Да, SQL-инструкция `INSERT` *отличается* от SQL-инструкции `SELECT`. Но, учитывая особенности использования запросов, в `log_request` не нужно блокировать работу приложения на время выполнения `INSERT`, а `view_the_log` должна дождаться результатов `SELECT`. Именно поэтому запросы *очень* различаются.

Это основное наблюдение.

Если `view_the_log` не дождется, пока `SELECT` вернет записи из базы данных, то код, следующий за `cursor.execute`, по всей видимости, завершится с ошибкой (поскольку не получит данные для обработки). Функция `view_the_log` **должна** быть блокирующей, поскольку она должна дождаться данных, *прежде чем* продолжит работу.

Вызывая `log_request`, веб-приложение просто хочет, чтобы функция записала детали текущего веб-запроса в базу данных. Для него не важно, *когда именно* это случится; важно, что это вообще будет сделано. Функция `log_request` ничего не возвращает; вызывающему коду нет необходимости ждать ответа. Для вызывающего кода важно, только чтобы веб-запрос *в конечном счете* был записан в журнал.

В связи с этим возникает вопрос: зачем `log_request` заставляет вызывающий код ждать?

Да. Это и есть наша безрассудная идея.

Когда пользователи веб-приложения вводят данные для нового поиска, их меньше всего заботит факт журналирования деталей запроса в какой-то базе данных, так что давайте не будем заставлять их ждать, пока веб-приложение делает эту работу.

Вместо этого организуем другой процесс, который *в конечном счете* будет производить журналирование независимо от основной функции веб-приложения (поиска букв в фразе).

Делаем несколько дел сразу

Вот план: организовать выполнение функции `log_request` независимо от основной функции веб-приложения. Для этого придется изменить код веб-приложения так, чтобы каждый вызов `log_request` запускался параллельно. И тогда веб-приложению больше не надо будет ждать завершения `log_request` перед обслуживанием другого запроса от другого пользователя (то есть больше никаких задержек).

Выполнится `log_request` мгновенно или потребуется несколько секунд, минута, даже часы — нашему веб-приложению все равно (как и всем нашим пользователям). Главное, чтобы этот код в конечном счете выполнился.

Параллельно выполняющийся код: есть варианты

Python поддерживает несколько способов параллельного выполнения некоторого кода в приложении. Помимо множества сторонних модулей, стандартная библиотека также включает некоторые механизмы, которые здесь пригодятся.

Среди них наиболее известной является библиотека `threading`, которая имеет высокоуровневый интерфейс к реализации многопоточности в операционной системе, где выполняется веб-приложение. Для использования библиотеки достаточно просто импортировать класс `Thread` из модуля `threading` в начале вашей программы.

```
from threading import Thread
```

Следуйте за нами и добавьте эту строку кода в начало файла `vsearch4web.py`.

Сейчас начнется веселье.

Чтобы получить новый поток выполнения, нужно создать объект `Thread`, передав в аргументе `target` имя функции для запуска в этом потоке и все необходимые ей аргументы в виде кортежа в другом именованном аргументе `args`. Созданный объект `Thread` затем присваивается выбранной вами переменной.

Полный список (и подробное описание) средств поддержки параллельного выполнения в стандартной библиотеке Python можно найти по адресу: https://docs.python.org/3/library/concurrency.html

В качестве примера предположим, что у нас есть функция с именем `execute_slowly`, которая принимает три аргумента (допустим, это три числа). Код, который вызывает `execute_slowly`, присваивает три значения переменным, называемым `glacial`, `plodding` и `leaden`. Вот как происходит нормальный вызов `execute_slowly` (то есть без всякого параллелизма):

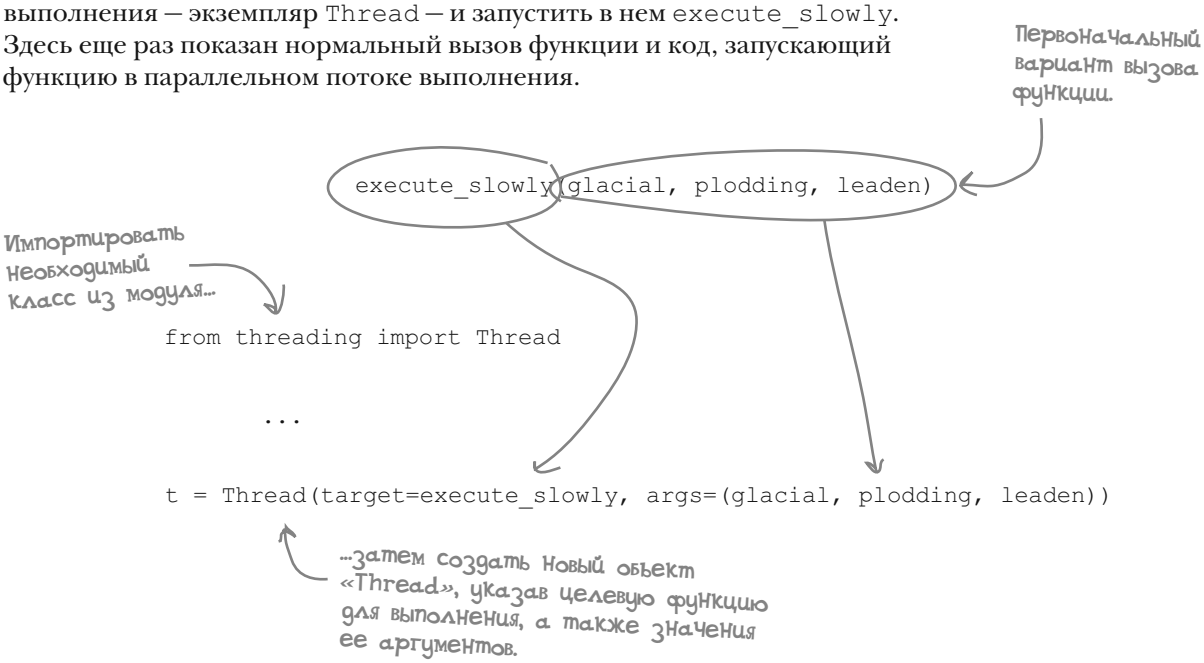
```
execute_slowly(glacial, plodding, leaden)
```

Если `execute_slowly` потребуется 30 секунд, чтобы сделать то, что она должна сделать, вызывающий код заблокируется и будет ждать 30 секунд, прежде чем сделает что-нибудь другое. Печально.

Не грустим: используем потоки

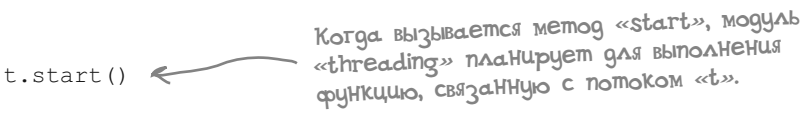
В глобальном масштабе 30 секунд ожидания до завершения функции `execute_slowly` — это не конец света. Но если пользователь сидит и ждет, он может решить, что, возможно, что-то пошло не так.

Если логика приложения позволяет продолжить работу, пока `execute_slowly` занимается своими делами, можно создать параллельный поток выполнения — экземпляр `Thread` — и запустить в нем `execute_slowly`. Здесь еще раз показан нормальный вызов функции и код, запускающий функцию в параллельном потоке выполнения.



Принятый порядок использования `Thread` выглядит немного странно, но в действительности это не так. Для понимания происходящего важно отметить, что объект `Thread` присваивается переменной (`t` в этом примере), а функции `execute_slowly` еще только предстоит выполниться.

Присваивание объекта `Thread` переменной `t` *подготавливает* его к выполнению. Чтобы потребовать от механизма многопоточности в Python выполнить `execute_slowly`, нужно запустить поток.



После этого код, вызвавший `t.start`, продолжит выполняться как обычно. 30-секундная задержка в `execute_slowly` не окажет никакого эффекта на вызывающий код, поскольку дальше `execute_slowly` выполняется уже под управлением модуля `threading`. Модуль `threading` сам договаривается с интерпретатором Python, когда *в конечном счете* выполнить `execute_slowly`.

Заточите карандаш

Теперь вернемся к вызову `log_request`. В веб-приложении этот вызов производится только в одном месте: в функции `do_search`. Как вы помните, мы уже заключили вызов `log_request` в `try/except` для защиты от неожиданных ошибок времени выполнения.

Заметьте также, что мы добавили 15-секундную задержку — используя `sleep(15)` — в код `log_request` (замедлив его). Вот как выглядит код `do_search` сейчас:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Так сейчас вызывается «log_request».

Мы предполагаем, что вы уже добавили инструкцию `from threading import Thread` в начало веб-приложения.

Возьмите карандаш и впишите в строки ниже код, который нужно вставить в `do_search` вместо стандартного вызова `log_request`.

Помните: вы должны запустить `log_request` с помощью объекта `Thread`, как в примере с `execute_slowly` на предыдущей странице.

Добавьте код для работы с потоками, который рано или поздно выполнит «log_request».

.....

.....

.....

Заточите карандаш
Решение

Мы вернулись к вызову `log_request`. В веб-приложении этот вызов производится только в одном месте: в функции `do_search`. Как вы помните, мы уже заключили вызов `log_request` в `try/except` для защиты от неожиданных ошибок времени выполнения.

Заметьте также, что мы добавили 15-секундную задержку — используя `sleep(15)` — в код `log_request` (замедлив его). Вот как выглядит код `do_search` сейчас:

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Так сейчас вызывается «log_request».

Мы предположили, что вы уже добавили инструкцию `from threading import Thread` в начало веб-приложения.

В пропуски ниже вы должны были вписать код, который нужно было вставить в `do_search` вместо стандартного вызова `log_request`.

Вам нужно было запустить `log_request` с помощью объекта `Thread`, как в примере с `execute_slowly`.

Сохраняем инструкцию «try» (пока).

```
try:
    t = Thread(target=log_request, args=(request, results))
    t.start()
except …
```

Как в примере выше, указываем целевую функцию для запуска, передаем аргументы, которые ей требуются, и не забываем запланировать запуск потока.

Блок «except» остается без изменений, поэтому мы его не показываем.

Пробная поездка

После всех этих изменений в `vsearch4web.py` можно приступать к тестированию. Сейчас мы надеемся избавиться от ожидания при вводе данных, пока веб-приложение выполняет поиск (так как `log_request` будет выполняться параллельно, под управлением модуля `threading`).

Двинемся вперед и проверим, как все работает.

Сразу после щелчка на кнопке «Do it!» веб-приложение вернуло результаты. Предположительно, прямо сейчас модуль `threading` выполняет `log_request` и ожидает, пока она завершится (приблизительно через 15 секунд).

И вы уже собирались похвалить себя (за хорошо сделанную работу), как вдруг — откуда ни возьмись — спустя 15 секунд окно терминала, где было запущено веб-приложение, выдало сообщения об ошибках.

Посмотрите на это сообщение.

Последний запрос был успешно обработан.

```
        ...
127.0.0.1 - - [29/Jul/2016 19:43:31] "POST /search4 HTTP/1.1" 200 -
Exception in thread Thread-6:
Traceback (most recent call last):
  File "vsearch4web.not.slow.with.threads.but.broken.py", line 42, in log_request
    cursor.execute(_SQL, (req.form['phrase'],
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
werkzeug/local.py", line 343, in __getattr__
        ...
    raise RuntimeError(_request_ctx_err_msg)
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/threading.py",
line 914, in _bootstrap_inner
    self.run()
        ...
RuntimeError: Working outside of request context.

This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem.
```

Упс! Неперехваченное исключение.

Еще куча !! сообщений об ошибках.

И еще... ой!

Если заглянуть в базу данных, можно заметить, что информация о веб-запросе **не** была записана. Судя по сообщениям выше, можно предположить, что модуль `threading` не вполне доволен вашим кодом. Множество сообщений интерпретатора из второй группы ссылаются на `threading.py`, в то время как первая группа сообщений ссылается на код в папках `werkzeug` и `flask`. Совершенно понятно, что добавление кода для работы с потоками приводит к **огромному беспорядку**. Что же произошло?

В первую очередь: не паниковать

Первое инстинктивное желание: откатить все изменения, добавленные для запуска `log_request` в отдельном потоке (и вернуться к рабочему состоянию). Но давайте не паниковать и **не** делать этого. Вместо этого посмотрим на следующий любопытный абзац, который появился дважды в сообщениях об ошибках.

```
...
This typically means that you attempted to use functionality that needed
an active HTTP request.  Consult the documentation on testing for
information about how to avoid this problem.
...
```

Обычно это означает, что вы попытались использовать функцию, требующую наличия активного HTTP-запроса. Чтобы узнать, как избежать этой проблемы, обращайтесь к документации по тестированию.

Это сообщение исходит не от модуля `threading`, а от Flask. Мы знаем это, потому что модулю `threading` все равно, он определенно не интересуется тем, что вы пытались что-то делать с HTTP.

Давайте заглянем в код, который должен выполняться в отдельном потоке и, как мы знаем, требует 15 секунд для работы, поскольку именно столько времени работает `log_request`. Пока вы рассматриваете этот код, подумайте, что могло случиться в течение этих 15 секунд?

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Что случится за 15 секунд, пока выполняется этот поток?

Поток немедленно будет запланирован для выполнения и вызывающий код (функция `do_search`) продолжит работу. Далее выполнится функция `render_template` (в мгновение ока) и функция `do_search` *завершится*.

Когда `do_search` завершается, все данные, связанные с функцией (ее *контекст*). удаляются интерпретатором. Переменные `request`, `phrase`, `letters`, `title` и `results` прекращают существование. Однако переменные `request` и `results` переданы в качестве аргументов в вызов `log_request`, которая попытается обратиться к ним 15 секунд спустя. К сожалению, к тому времени переменные будут удалены, так как `do_search` завершилась. Печально.

Не грустим: Flask нам поможет

Опираясь на новые знания, можно заключить, что функция `log_request` (когда выполняется в отдельном потоке) не «видит» своих аргументов. Это произошло из-за того, что интерпретатор уже давно прибрал за собой и очистил память, занимаемую этими переменными (поскольку `do_search` завершилась). В частности, объект `request` больше не активен, и попытка функции `log_request` найти его терпит неудачу.

Итак, что можно сделать? Не волнуйтесь: помощь близко.

На самом деле не надо ничего переписывать.

На первый взгляд может показаться, что нам нужно переписать `log_request`, чтобы в меньшей мере полагаться на его аргументы... подразумевая, что это даже возможно. Но оказывается, в фреймворке Flask есть декоратор, который может помочь в этом случае.

Декоратор `copy_current_request_context` гарантирует, что HTTP-запрос, который активен в момент вызова функции, *останется* активным, когда функция позже запустится в отдельном потоке. Чтобы воспользоваться им, нужно добавить `copy_current_request_context` в список импорта в начале веб-приложения.

Как любой другой декоратор, он применяется к существующей функции с использованием обычного `@`-синтаксиса. Но будьте внимательны: декорируемая функция должна быть определена *внутри* функции, которая ее вызывает; то есть она должна быть вложена в вызывающую ее функцию.

Упражнение

Мы хотим, чтобы вы выполнили следующее (после обновления списка импорта из Flask).

1. Перенесите функцию `log_request` внутрь функции `do_search`.
2. Декорируйте `log_request` с помощью `@copy_current_request_context`.
3. Проверьте, исчезли ли ошибки, наблюдавшиеся в последней «Пробной поездке».

Упражнение Решение

Мы просили вас сделать следующее:

1. Перенести функцию `log_request` внутрь функции `do_search`.
2. Декорировать `log_request` с помощью `@copy_current_request_context`.
3. Убедиться, что ошибки, наблюдавшиеся в последней «Пробной поездке», пропали.

Вот как выглядит код `do_search` после того, как мы выполнили пункты 1 и 2 (обратите внимание: пункт 3 обсуждается на следующей странице).

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15)  # This makes log_request really slow...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                    (phrase, letters, ip, browser_string, results)
                    values
                    (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Задание 2. Перед «log_request» добавлен декоратор.

Задание 1. Функция «log_request» сейчас определена (вложена) внутри функции «do_search».

Весь остальной код без изменений.

Это не Глупые вопросы

В: Имеет ли смысл защищать вызов log_request в отдельном потоке с помощью try/except?

О: Нет, если вы надеетесь обработать ошибки времени выполнения в `log_request`, потому что `try/except` завершится прежде, чем запустится поток. Однако вашей системе может не удаться создать новый поток, поэтому стоит оставить `try/except` в `do_search`.

Пробная поездка

Задание 3. Опробование последней версии `vsearch4web.py` подтверждает, что ошибки времени выполнения из предыдущей «Пробной поездки» теперь в прошлом. Окно терминала, где запущено веб-приложение, подтверждает, что все хорошо.

```
    ...
127.0.0.1 - - [30/Jul/2016 20:42:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:10] "POST /search4 HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:14] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:17] "GET /viewlog HTTP/1.1" 200 -
127.0.0.1 - - [30/Jul/2016 20:43:37] "GET /viewlog HTTP/1.1" 200 -
```

Страшные исключения времени выполнения исчезли. Все эти «200» означают, что с веб-приложением все хорошо. И через 15 секунд после отправки нового поискового запроса веб-приложение наконец записывает детали в базу данных, НЕ заставляя пользователя ждать. ☺

Согласно карточке, я могу задать последний вопрос. Есть ли недостатки в том, что мы определили «log_request» внутри «do_search»?

Нет. Не в нашем случае.

В этом веб-приложении функция `log_request` всегда вызывалась только внутри `do_search`, так что перенос `log_request` внутрь `do_search` не является проблемой.

Если позднее вы решите вызывать `log_request` из какой-то другой функции, это может стать проблемой (и вам придется кое-что переосмыслить). Но прямо сейчас вы великолепны.

Надежно ли ваше веб-приложение сейчас?

Вот четыре вопроса, сформулированные в начале главы 11.

1. **Что случится, если попытка подключения к базе данных потерпит неудачу?**
2. **Защищено ли наше веб-приложение от веб-атак?**
3. **Что произойдет, если что-нибудь потребует много времени?**
4. **Что случится, если вызов функции потерпит неудачу?**

Сейчас наше веб-приложение обрабатывает множество исключений времени выполнения, благодаря использованию `try/except`, а также некоторые пользовательские исключения, которые можно возбуждать и перехватывать по мере необходимости.

Если известно, что во время выполнения что-нибудь может пойти не так, защитите свой код от любых исключений, которые могут случиться. Это улучшит общую надежность вашего приложения, что очень хорошо.

Обратите внимание, что в приложении еще остался код, надежность которого можно улучшить. Мы потратили много времени, добавляя код `try/except` в функцию `view_the_log`, которая использует диспетчера контекста `UseDatabase`. Но `UseDatabase` *также* используется внутри `log_request`, и ее тоже следовало бы защитить (оставляем это вам в качестве упражнения для домашней работы).

Наше веб-приложение стало реагировать быстрее, благодаря переносу в параллельный поток задачи, которая должна быть выполнена, но необязательно сразу. Это удачное архитектурное решение, но будьте осторожны и не злоупотребляйте потоками: пример организации многопоточного выполнения в этой главе очень прост. Однако очень легко создать многопоточный код, который никто не сможет понять и который будет сводить вас с ума во время отладки. **Используйте потоки с осторожностью.**

Теперь вернемся к вопросу 3: *что случится, если что-нибудь потребует много времени?* Использование потоков улучшает производительность записи в базу данных, но не чтения. Это тот случай, когда не остается ничего иного, кроме как ждать получения данных после отправки запроса, сколько бы времени это ни заняло, так как без этих данных веб-приложение не способно продолжить работу.

Чтобы ускорить чтение из базы данных (если оно на самом деле выполняется медленно), можно подумать об использовании альтернативных (более быстрых) баз данных. Но это совсем другая тема, и мы не будем ее рассматривать.

Тем не менее в следующей, последней главе мы исследуем проблемы производительности, но сделаем это в процессе обсуждения итераций — темы, которую все понимают и которую мы уже обсуждали.

Код из главы 11¾, 1 из 2

Последняя и лучшая версия «vsearch4web.py».

```
from flask import Flask, render_template, request, escape, session
from flask import copy_current_request_context
from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLError
from checker import check_logged_in

from threading import Thread
from time import sleep

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15)  # This makes log_request really slow...
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """insert into log
                    (phrase, letters, ip, browser_string, results)
                    values
                    (%s, %s, %s, %s, %s)"""
            cursor.execute(_SQL, (req.form['phrase'],
                                  req.form['letters'],
                                  req.remote_addr,
                                  req.user_agent.browser,
                                  res, ))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
```

Оставшаяся часть «do_search» — в начале следующей страницы. →

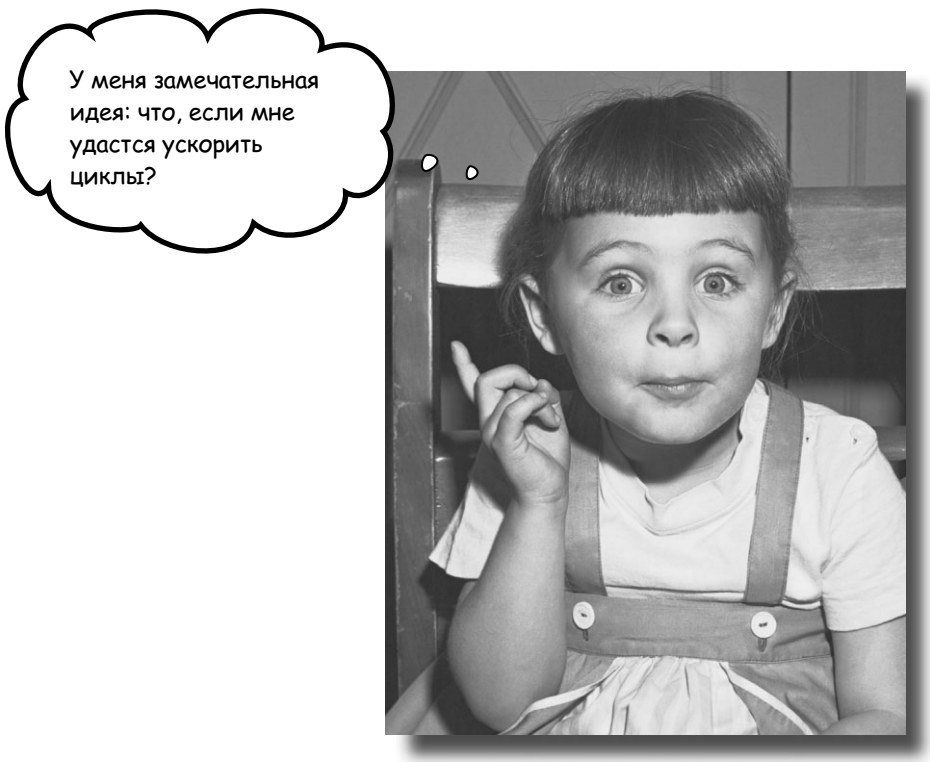
Код из главы 11¾, 2 из 2

```
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                    from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Some unknown exception.")
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                               the_title='View Log',
                               the_row_titles=titles,
                               the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except SQLError as err:
        print('Is your query correct? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

Оставшаяся часть функции «do_search».

12 продвинутые итерации

Безумные циклы



Удивительно, сколько времени порой наши программы проводят внутри циклов.

Ни для кого не секрет, что большинство программ предназначено для быстрого выполнения одинаковых действий много-много раз. Говоря об оптимизации циклов, можно выделить два подхода: (1) улучшение синтаксиса (чтобы упростить определение циклов) и (2) улучшение способа выполнения (чтобы ускорить работу циклов).

Во времена Python 2 (то есть давным-давно) создатели языка добавили единственную языковую возможность, которая реализует оба подхода и называется довольно странно — **генераторы**. Но пусть это название вас не пугает. К концу главы вы будете удивляться, как вам вообще удавалось столько времени обходиться без генераторов.

Компании Bahamas Buzzers есть что предложить

Мы всегда готовы предложить, куда слетать и что посмотреть...

Чтобы понять, что могут предложить генераторы, рассмотрим порцию реальных данных.

Работая в Нассау на острове Нью-Провиденс, компания *Bahamas Buzzers* организует перелеты в аэропорты наиболее крупных островов. Авиакомпания использует передовую схему оперативного планирования рейсов: на основе спроса за предыдущий день авиакомпания способна предсказать (просто синоним слова «угадать»), сколько рейсов понадобится на следующий день. В конце каждого дня в офисе *Bahamas Buzzers* составляют расписание полетов на следующий день и сохраняют в виде текстового файла в формате CSV (Comma-Separated Values — значения, разделенные запятыми).

Вот что включает в себя файл CSV с расписанием на завтра:

```
TIME,DESTINATION
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

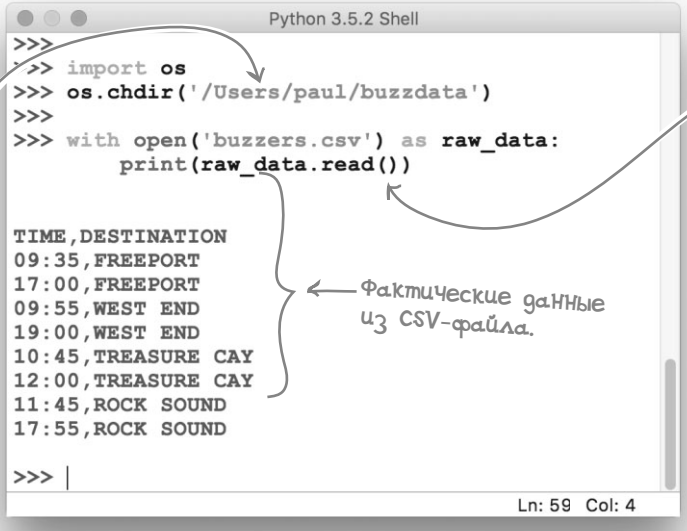
Заголовок говорит, что есть колонки: одна представляет время, другая — направление.

Оставшаяся часть CSV-файла — собственно расписание полетов.

Это стандартный CSV-файл. Первая строка содержит заголовки. Все вроде бы нормально, кроме того, что все заголовки записаны только буквами ВЕРХНЕГО РЕГИСТРА (что немного «отдает стариной»).

В офисе *Bahamas Buzzers* этот файл называют `buzzers.csv`.

Если бы вам потребовалось прочитать данные из CSV-файла и отобразить их на экране, вы наверняка использовали бы инструкцию `with`. Вот что мы сделали в командной строке IDLE, после того как с помощью модуля `os` переместились в папку, содержащую этот файл.



Папка, где находится требуемый файл.

Метод «read» читает все символы из файла за один прием.

Фактические данные из CSV-файла.

Для умников

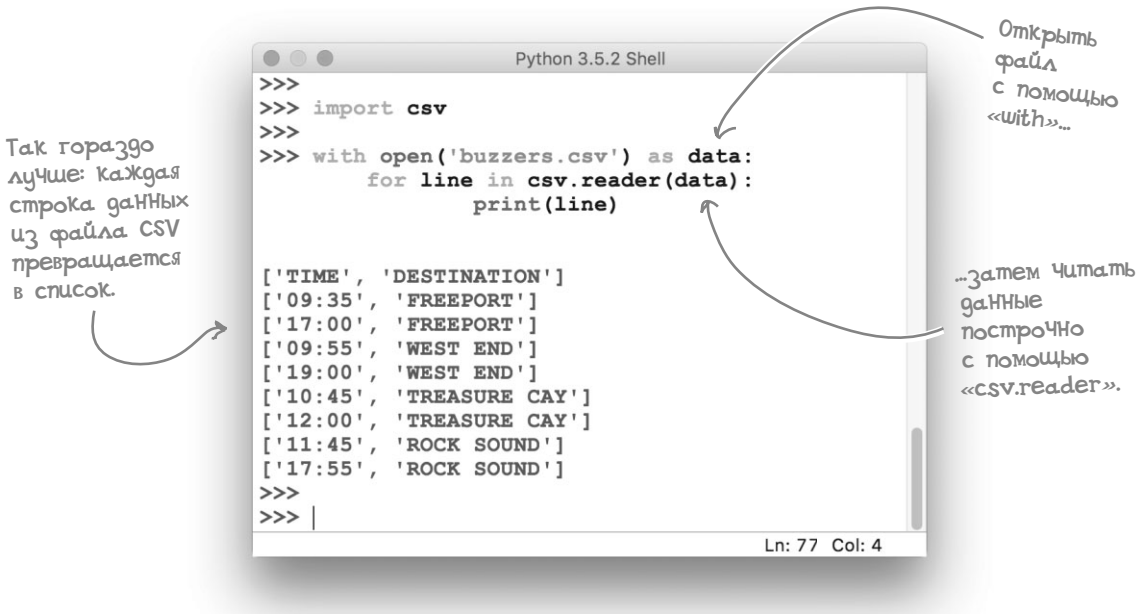
Узнайте больше о формате CSV здесь: *https://en.wikipedia.org/wiki/Comma-separated_values* (англ.) или *https://ru.wikipedia.org/wiki/CSV* (рус.).

Чтение данных CSV в виде списков

Обычные CSV-данные в необработанном виде не очень полезны. Гораздо практичнее читать строки по отдельности и разбивать их по разделителю-запятой, чтобы упростить извлечение данных.

Хотя такое «разбиение» можно реализовать на Python вручную (с помощью метода `split` строковых объектов), работать с данными в формате CSV приходится настолько часто, что в *стандартную библиотеку* был добавлен модуль `csv`, способный помочь в этом.

Вот еще один небольшой цикл `for`, демонстрирующий модуль `csv` в действии. В отличие от последнего примера, где использовался метод `read`, читающий содержимое файла *в один прием*, в следующем фрагменте используется метод `csv.reader`, который читает файл CSV построчно, внутри цикла `for`. В каждой итерации в теле цикла `for` строка CSV-данных присваивается переменной с именем `line` и затем выводится на экран.



Основную работу здесь выполняет модуль `csv`. Он читает из файла исходные данные построчно, а затем «как по волшебству» превращает строки в списки из двух элементов.

Кроме информации из заголовка (из первой строки в файле), которая возвращается в виде списка, каждая пара со временем вылета и пункта назначения также становится отдельным списком. Обратите внимание на *тип* данных возвращаемых элементов: это строковые значения, даже притом что первый элемент списка представляет время.

Модуль `csv` прячет много сюрпризов в рукаве. Еще одна интересная функция — `csv.DictReader`. Давайте посмотрим, что она делает.

Чтение CSV-данных в словарь

Следующий пример похож на предыдущий, но в нем вместо `csv.reader` используется `csv.DictReader`. Функция `DictReader` возвращает данные из CSV-файла в виде коллекции словарей, в которых роль ключей играют строки из заголовка файла, а значения извлекаются из последующих строк. Вот этот код.

Чтобы задействовать «csv.DictReader», достаточно простого изменения, но оно дает ощутимый результат. То, что было списками (в прошлый раз), стало теперь словарями.

```
Python 3.5.2 Shell
>>>
>>>
>>> with open('buzzers.csv') as data:
        for line in csv.DictReader(data):
            print(line)

{'DESTINATION': 'FREEPORT', 'TIME': '09:35'}
{'DESTINATION': 'FREEPORT', 'TIME': '17:00'}
{'DESTINATION': 'WEST END', 'TIME': '09:55'}
{'DESTINATION': 'WEST END', 'TIME': '19:00'}
{'DESTINATION': 'TREASURE CAY', 'TIME': '10:45'}
{'DESTINATION': 'TREASURE CAY', 'TIME': '12:00'}
{'DESTINATION': 'ROCK SOUND', 'TIME': '11:45'}
{'DESTINATION': 'ROCK SOUND', 'TIME': '17:55'}
>>>
>>>
                                    Ln: 79  Col: 19
```

Ключи.

Значения.

```
TIME,DESTINATION
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Как вы помните: необработанные данные из файла выглядят так.

Нет сомнений, это очень мощный инструмент: одним вызовом `DictReader` модуль `csv` трансформирует данные из CSV-файла в коллекцию словарей Python.

Но представьте, что вам дали задание преобразовать исходные данные из файла CSV в соответствии со следующими требованиями.

1. **Преобразовать время вылета из 24-часового формата в 12-часовой.**
2. **Преобразовать буквы в названиях пунктов назначения из ВЕРХНЕГО РЕГИСТРА в Регистр Заголовка.**

В регистре заголовка каждое слово начинается с заглавной буквы (Великие Луки, Нижний Новгород, Минеральные Воды).

Сами по себе требования не выглядят сложными. Однако учитывая, что исходные данные хранятся в виде коллекции списков или словарей, они могут показаться трудными. Поэтому давайте напишем собственный цикл `for` для чтения данных в один словарь, который потом будет проще использовать для этих преобразований.

Вернемся немного назад

Вместо применения функций `csv.reader` или `csv.DictReader` мы напишем свой код, чтобы прочитать данные из CSV-файла в *единый* словарь, который потом будем использовать для необходимых преобразований.

Мы пообщались в чате с представителями офиса *Bahamas Buzzers*, и они сказали, что хотели бы получить все задуманные нами преобразования, но все же предпочитают хранить данные в «исходном виде», потому что их устаревшее табло с расписанием может получать данные только в таком формате: время вылета в 24-часовом формате и название пункта назначения из букв в ВЕРХНЕМ РЕГИСТРЕ.

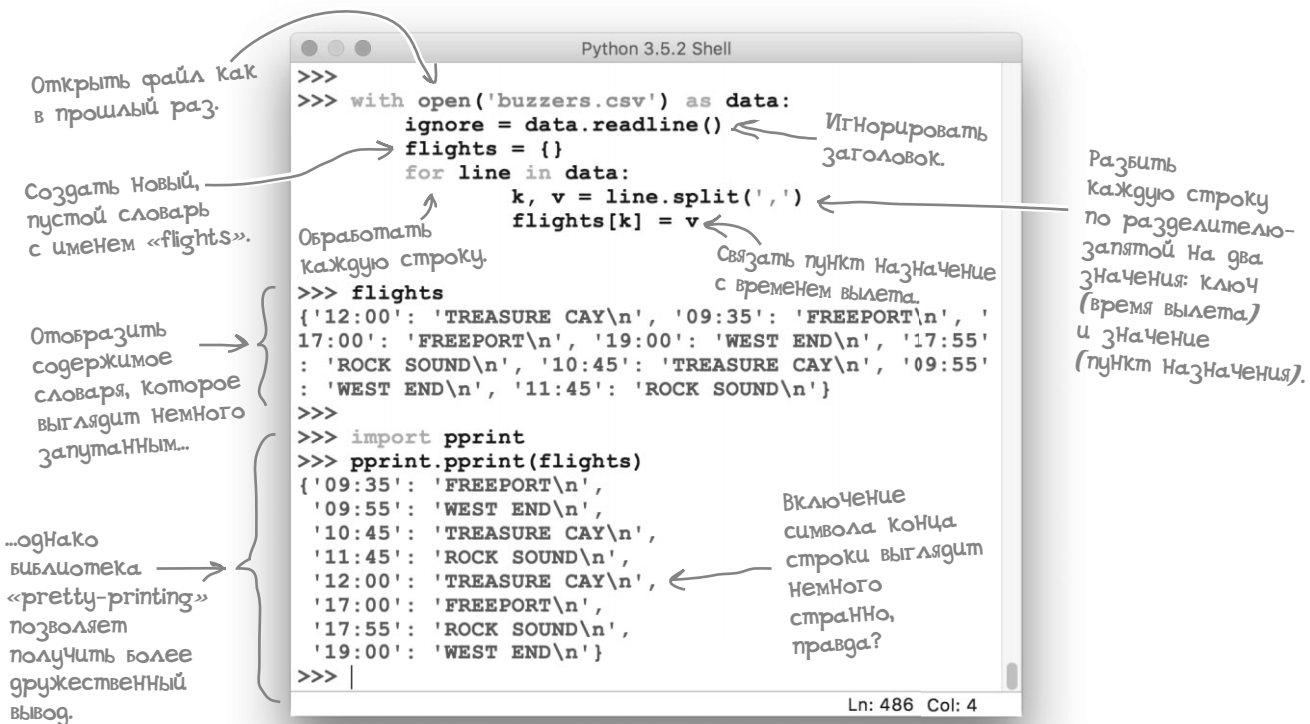
Мы можем выполнить преобразование данных в своем словаре, но будем использовать для этого *копию*, а не фактические исходные данные, прочитанные из файла. Пока до конца не ясно, но из офиса доходят слухи, что вдобавок к нашему коду потребуется реализовать интерфейс с некоторыми существующими системами. Поэтому, чтобы потом не пришлось преобразовывать данные обратно в исходную форму, будем читать данные в отдельный словарь в том виде, какой есть, а затем преобразовывать их копию так, как требуется (а исходные данные в оригинальном словаре пусть останутся *нетронутыми*).

Исходные данные.

```
TIME,DESTINATION
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Сможете разбить каждую строку, используя запятую как разделитель?

Чтение исходных данных в словарь не требует особых усилий (сверх тех, что потребовалось приложить при использовании модуля `csv`). Код ниже открывает файл, читает и отбрасывает первую строку (потому что нам не нужна информация о заголовке). Следующий далее цикл `for` читает каждую строку с исходными данными, разбивает ее на две по разделителю-запятой и использует время вылета как *ключ* в словаре, а пункт назначения — как *значение*.



Удаление пробелов и разбивка исходных данных

В последней инструкции `with` использовался метод `split` (которым обладают все строковые объекты), чтобы разбить исходную строку на две. Он возвращает список строк, которые тут же присваиваются переменным `k` и `v`. Такое присваивание нескольким переменным возможно, потому что слева от оператора присваивания у нас находится кортеж переменных, а справа — код, возвращающий список значений (помните: кортежи — это *неизменяемые* списки):

```
...
k, v = line.split(',')
...
```

Кортеж переменных слева.

Код, возвращающий список значений, справа.

Другой строковый метод, `strip`, удаляет пробельные символы в начале и в конце строки. Используем его, чтобы убрать нежелательные символы конца строки из исходных данных *перед* вызовом `split`.

Вот окончательная версия кода для чтения данных. Мы создали словарь `flights`, в котором время вылета используется как ключи, а пункты назначения (без символа конца строки) — как значения.

Для умников

Пробельные символы: пробельными считаются следующие символы: *пробел*, `\t`, `\n` и `\r`.

```
Python 3.5.2 Shell
>>>
>>> with open('buzzers.csv') as data:
        ignore = data.readline()
        flights = {}
        for line in data:
                k, v = line.strip().split(',')
                flights[k] = v


>>>
>>> pprint.pprint(flights)
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
>>>
>>>
Ln: 575 Col: 4
```

Этот код удаляет пробельные символы из строки, а затем разбивает ее, чтобы получить данные в требуемом формате.

Возможно, вы не заметили, но порядок строк в словаре отличается от порядка элементов в файле. Это происходит потому, что словари НЕ сохраняют порядок добавления. Пока не беспокойтесь об этом.

```
TIME,DESTINATION
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Что, если поменять порядок вызова методов вот так:

```
line.split(',').strip()
```

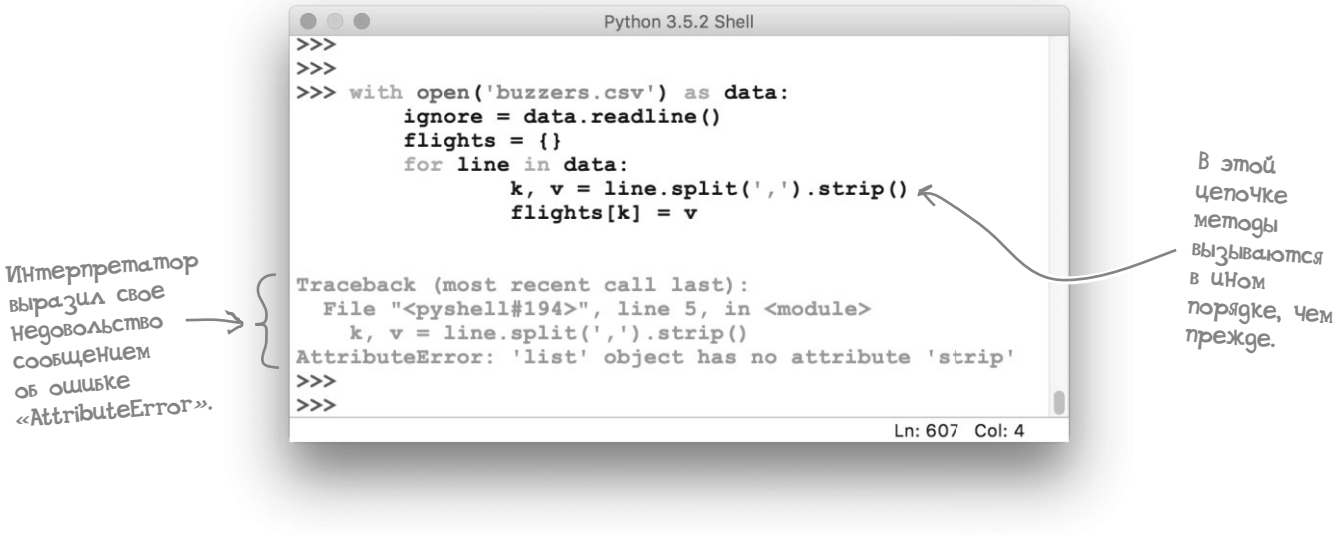
Такой способ последовательного вызова методов называется «цепочкой методов».

Как вы думаете, что произойдет?

Будьте внимательны при составлении цепочек из вызовов методов

Некоторые программисты не пользуются возможностью объединения вызовов методов в цепочки (как `strip` и `split` в предыдущем примере), потому что такие цепочки трудно читать с непривычки. Однако прием объединения вызовов в цепочки пользуется популярностью среди программистов на Python, поэтому вы наверняка будете встречаться с этим приемом «в дикой природе». Осторожность, однако, не помешает, потому что порядок вызова методов *имеет значение.*

Чтобы убедиться в этом, рассмотрим следующий код (очень похожий на то, что мы видели раньше). В прошлый раз первым вызывался `strip`, а затем `split`, в этом коде сначала вызывается `split`, а затем `strip`. Посмотрим, что получилось.



Чтобы понять, что произошло, посмотрим, как изменяется *тип* данных справа от оператора присваивания во время выполнения цепочки методов.

Прежде чем что-либо произойдет, переменная `line` сохранит строку. Метод `split` разбивает строку, используя свой аргумент как разделитель, и возвращает список строк. Объект, первоначально бывший *строкой* (`line`), динамически трансформировался в *список* и затем выполняется попытка вызвать следующий метод. В нашем примере следующий метод — `strip`. Однако списки не имеют такого метода, поэтому интерпретатор генерирует исключение `AttributeError`.

Цепочка методов на предыдущей странице не страдает этим недостатком:

```
...
line.strip().split(',')
...
```

В этом коде интерпретатор начинает со строки (`line`), из которой с помощью метода `strip` удаляет начальные/конечные пробельные символы (получая другую строку), а затем результат разбивается с помощью `split` на список строк по разделителю-запятой. Нет никакой ошибки `AttributeError`, потому что в цепочке методов не нарушены правила использования типов.

Преобразование данных в нужный формат

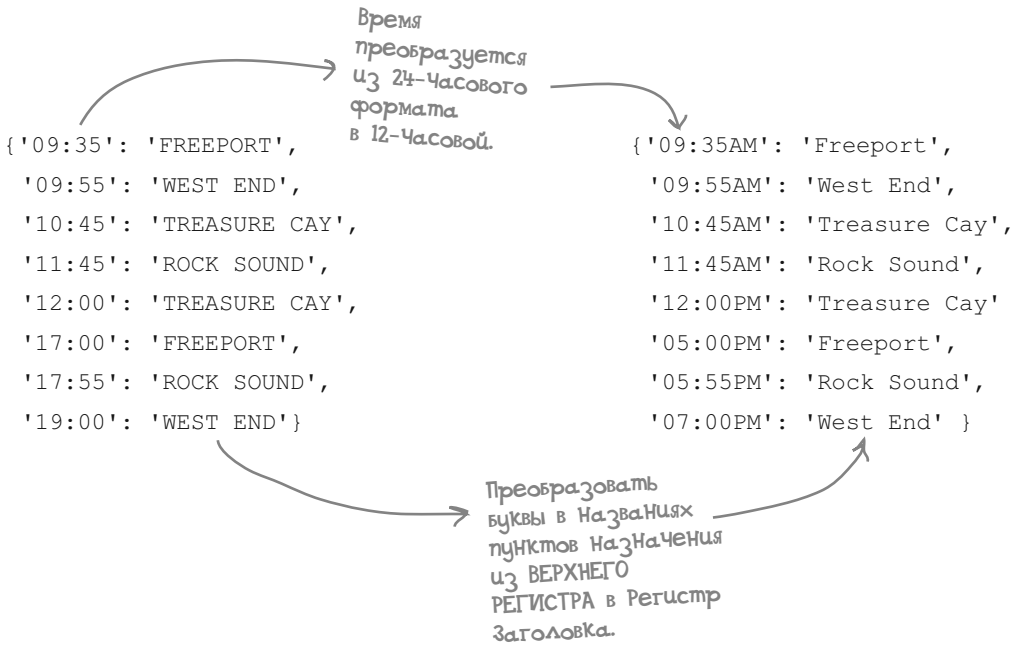
Сейчас, когда все данные собраны в словарь `flights`, давайте подумаем, как их преобразовать, чтобы привести в соответствие с пожеланиями авиакомпании *BB*.

Нам нужно произвести следующие два преобразования, оговоренные ранее в главе, попутно создав новый словарь.

1. **Преобразовать время вылета из 24-часового формата в 12-часовой.**

2. **Преобразовать буквы в названиях пунктов назначения из ВЕРХНЕГО РЕГИСТРА в Регистр Заголовка.**

Применение этих преобразований к словарю `flights` позволит превратить словарь слева в словарь справа.

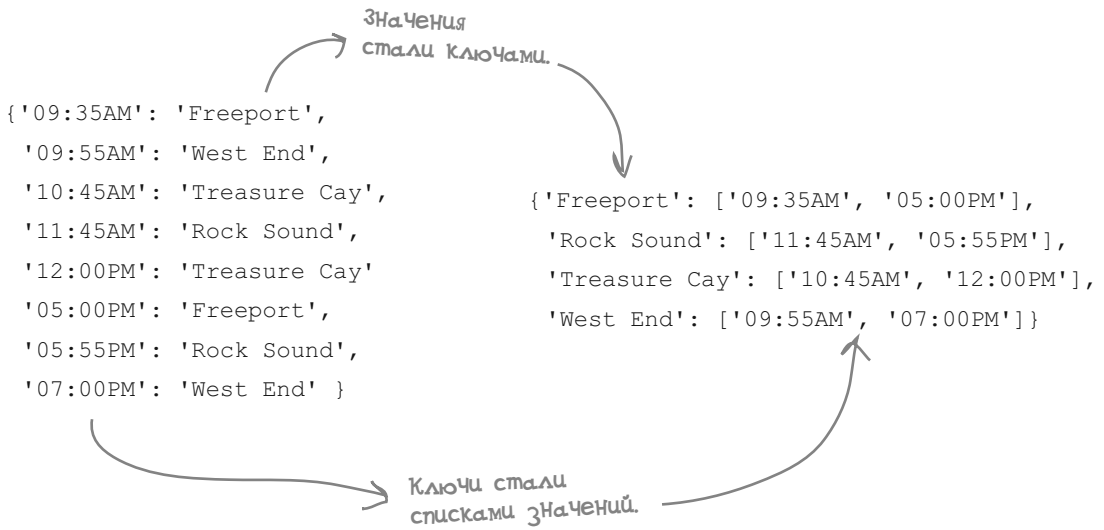


Обратите внимание: оба словаря содержат одни и те же значения, меняется только форма представления. Второй словарь нужен в офисе авиакомпании, потому что так данные выглядят понятнее и привычнее; в офисе считают ВЕРХНИЙ РЕГИСТР сродни крику.

Сейчас оба словаря хранят все комбинации времени вылета/пункта назначения в виде отдельных элементов. Хотя в офисе авиакомпании будут довольны уже тем, что словарь слева превратится в словарь справа, они полагают, было бы полезно сгруппировать данные так, чтобы название пункта назначения играло роль ключа, а список с временами вылета в этот пункт — роль значения, то есть по одной записи для каждого пункта назначения. Посмотрим, как мог бы выглядеть *такой* словарь, прежде чем начинать манипуляции с кодом.

Преобразование в словарь списков

Помимо преобразования данных в словаре `flights`, авиакомпания предложила произвести еще одну манипуляцию (обсуждалась в конце предыдущей страницы).



Подумаем о порядке преобразования данных...

Чтобы прочитать исходные данные из CSV-файла и получить словарь списков, который показан выше справа, требуется приложить некоторые усилия. Давайте подумаем, как можно это сделать с помощью уже известных нам приемов в Python.

Большинство программистов быстро догадаются, что в данном случае поможет цикл `for`. Как главный механизм организации циклов в Python, `for` уже помог нам получить исходные данные из файла CSV и наполнить ими словарь `flights`.

Это классический способ использования «for» и очень популярная идиома программирования на Python.

```
with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v
```

Велик соблазн изменить этот код так, чтобы он преобразовывал исходные данные из CSV-файла по мере их чтения, то есть *перед* добавлением элемента в словарь `flights`. Но, как вы помните, в *авиакомпании* хотели бы, чтобы исходные данные в словаре `flights` остались нетронутыми: любые преобразования должны применяться к **копии** данных. Это осложняет нам работу, но ненамного.

Выполняем простые преобразования

На данный момент словарь `flights` хранит времена вылета в 24-часовом формате в виде ключей, а названия пунктов назначения, состоящие из букв В ВЕРХНЕМ РЕГИСТРЕ, — в виде значений. Сначала нам нужно выполнить два преобразования.

1. **Преобразовать время вылета из 24-часового формата в 12-часовой.**

2. **Преобразовать буквы в названиях пунктов назначения из ВЕРХНЕГО РЕГИСТРА в Регистр Заголовка.**

Преобразование № 2 очень простое, выполним его первым. Для строковых данных достаточно вызвать метод `title`, как показано в следующем сеансе IDLE.

Метод «title» возвращает копию данных из «s».

```
>>> s = "I DID NOT MEAN TO SHOUT."
>>> print(s)
I DID NOT MEAN TO SHOUT.
>>> t = s.title()
>>> print(t)
I Did Not Mean To Shout.
```

Так выглядит значительно лучше, чем до этого.

Преобразование № 1 требует немного больше работы.

Если подумать как следует, становится понятным, что преобразование, например `19:00` в `7:00PM`, потребует немалых усилий. Чтобы его выполнить, придется написать довольно много кода. Однако это касается только случая, когда значение `19:00` представлено строкой.

Если, напротив, рассматривать значение `19:00` как время, можно воспользоваться преимуществами модуля `datetime`, который входит в *стандартную библиотеку* Python. Класс `datetime`, определяемый в этом модуле, может принять строку (как `19:00`) и преобразовать ее в эквивалентный 12-часовой формат с помощью двух встроенных функций и так называемых *спецификаторов формата*. Вот маленькая функция `convert2ampm`, которая использует возможности модуля `datetime` для нужного нам преобразования.

Более подробную информацию о спецификаторах формата вы найдете по адресу: https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior

Готовый код

```
from datetime import datetime

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')
```

Получая время в 24-часовом формате (в виде строки), данная цепочка методов преобразует его в строку в 12-часовом формате.

Заточите карандаш

Давайте используем прием преобразования, рассмотренный на предыдущей странице.

Код внизу читает исходные данные из CSV-файла и заполняет словарь `flights`. Также показана функция `convert2ampm`.

Ваша задача — написать цикл `for`, который извлекает данные из `flights` и преобразует ключи в 12-часовой формат, а значения — в *Регистр Заголовка*. Для хранения преобразованных данных создается новый словарь с именем `flights2`. В пропуски впишите карандашом код цикла `for`.

Подсказка: работая со словарем в цикле `for`, помните, что в каждой итерации метод `items` возвращает ключ и значение каждой записи (в виде кортежа).

Объявление функции преобразования.

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}

.....

.....

pprint.pprint(flights2)
```

Извлечь данные из файла.

Форматированный вывод содержимого словаря «flights» перед началом преобразований.

Новый словарь с именем «flights2», первоначально пустой.

Добавьте сюда свой цикл «for».

Форматированный вывод содержимого словаря «flights2», чтобы убедиться в правильности преобразований.

Заточите карандаш
Решение

Ваша задача состояла в том, чтобы написать цикл `for`, извлекающий данные из `flights` и преобразующий ключи в 12-часовой формат, а значения — в *Регистр Заголовка*. Вам нужно было вписать в пропуски код цикла `for`, который сохраняет преобразованные данные в новый словарь `flights2`.

Мы сохранили код в файле «do_convert.py».

```
from datetime import datetime
import pprint


def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')


with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()
pprint.pprint(flights2)
```

Метод «items» возвращает элементы словаря «flights» по одному.

В каждой итерации ключ (в переменной «k») преобразуется в 12-часовой формат, а затем используется в качестве ключа в новом словаре.

Значение (в переменной «v») преобразуется в Регистр Заголовка, а затем присваивается преобразованному ключу.

Пробная поездка

Если запустить программу выше, на экране появятся два словаря (показаны чуть ниже). Преобразование выполняется успешно, но порядок следования записей в словарях отличается, потому что при наполнении нового словаря данными интерпретатор **не** сохраняет *порядок добавления*.

Это «flights».

```
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
```

```
{'05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End',
 '09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'}
```

Это «flights2».

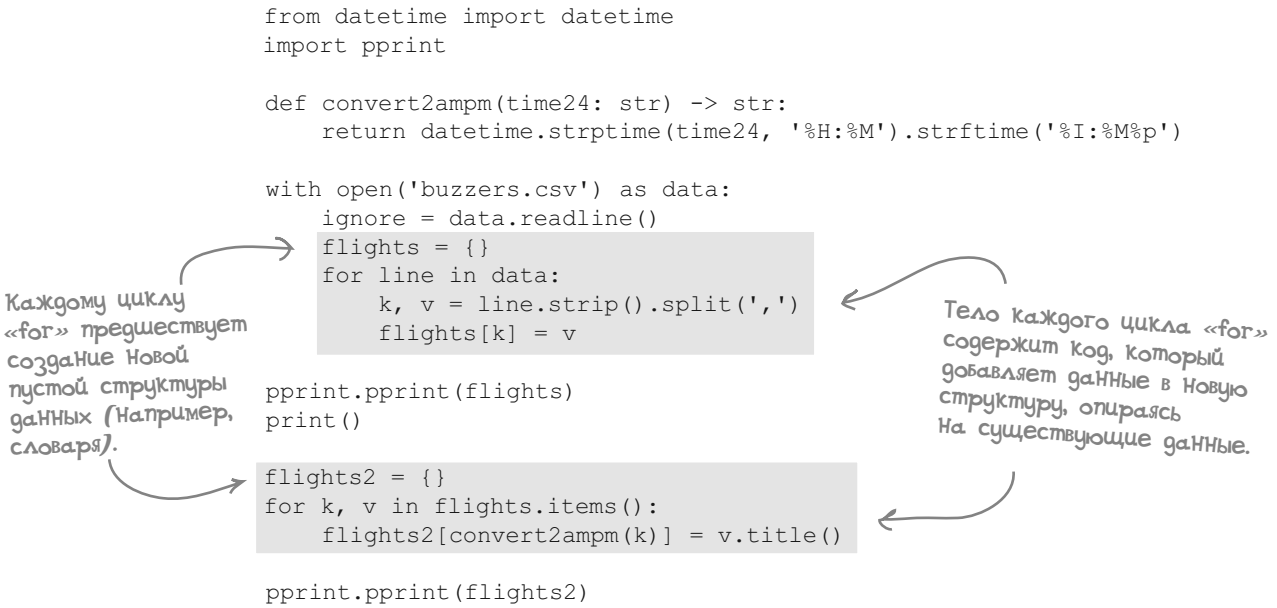
Исходные данные преобразованы.

Вы заметили шаблон в коде?

Рассмотрим только что опробованную программу под другим углом. В ней *дважды* используется довольно распространенный шаблон. Заметили его?

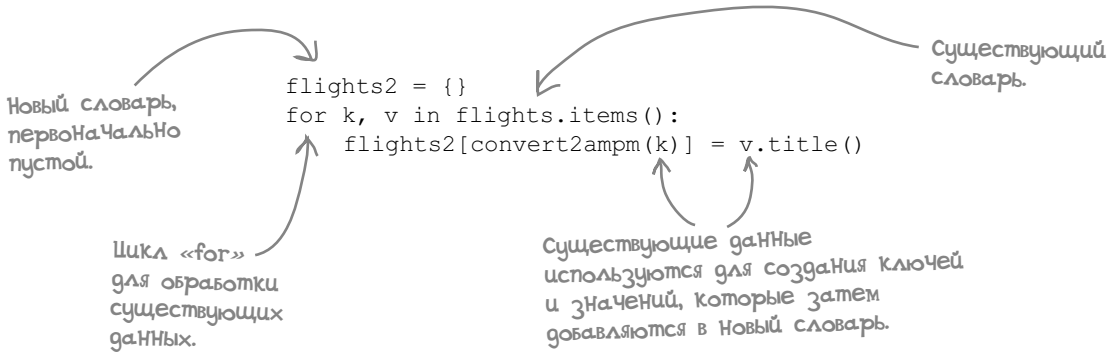
```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()

pprint.pprint(flights2)
```

Если вы ответили: «цикл `for`», то правы только наполовину. Цикл `for` — часть этого шаблона, но взгляните еще раз на *окружающий* его код. Заметили еще что-нибудь?

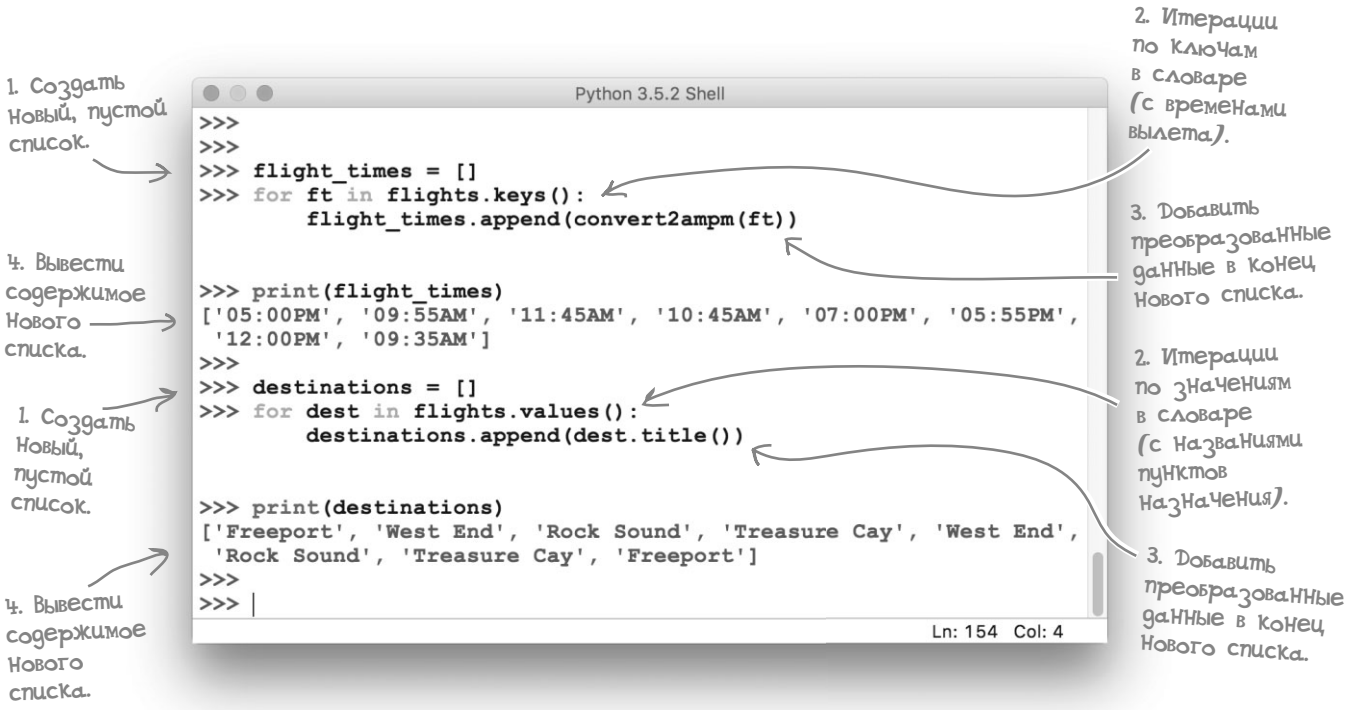


Шаблон со списками

Примеры на предыдущей странице иллюстрируют применение распространенного шаблона программирования со словарями: сначала создается новый, пустой словарь, потом в цикле `for` происходит обработка существующего словаря и создание данных для нового.



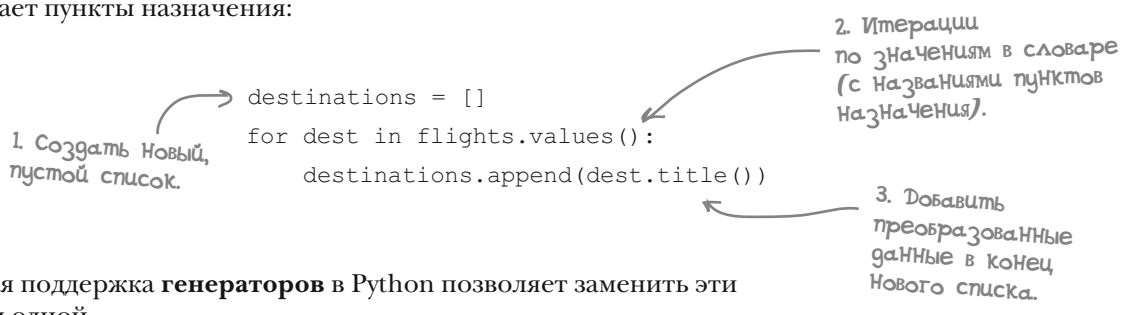
Этот шаблон также используется со списками, из-за чего становится еще заметнее. Взгляните на листинг сеанса в IDLE, где ключи (время вылета) и значения (пункты назначения) извлекаются из словаря `flights` в виде списков, а затем преобразуются в новые списки, с использованием уже известного шаблона (его этапы пронумерованы в аннотациях от 1 до 4).



Этот шаблон используется так часто, что в Python была добавлена удобная сокращенная форма его записи — **генератор**. Сейчас мы посмотрим, как определяются генераторы.

Преобразование шаблонного кода в генераторы

Рассмотрим в качестве примера последнюю версию цикла `for`, которая обрабатывает пункты назначения:

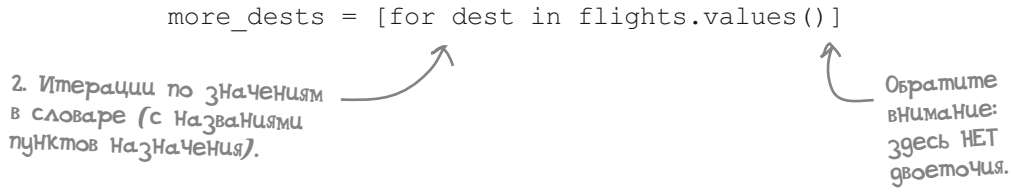


Встроенная поддержка **генераторов** в Python позволяет заменить эти три строки одной.

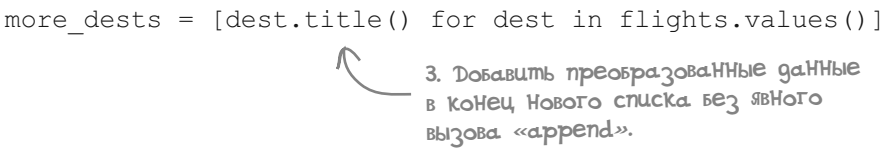
Чтобы заменить три строки кода выше одной, мы шаг за шагом пройдем процесс создания законченного генератора.

Сначала создадим новый, пустой список, который присвоим переменной (в нашем случае `more_dests`).

Определим, как будут осуществляться итерации по существующим данным (`flights` в нашем примере), с использованием уже знакомой нам нотации `for` и поместим этот код внутрь квадратных скобок, обозначающих создание нового списка (обратите внимание на *отсутствие* двоеточия в конце инструкции `for`).



Чтобы завершить создание генератора, нужно определить преобразование, применяемое к данным (в `dest`), и поместить это преобразование *перед* ключевым словом `for` (обратите внимание на *отсутствие* вызова метода `append` — генератор вызывает его автоматически).



Вот и все. Единственная строка кода в конце страницы функционально эквивалентна трем строкам в ее начале. Продолжим работу и выполним код в командной строке >>>, чтобы убедиться, что список `more_dests` содержит те же данные, что и список `destinations`.

Подробнее о генераторах

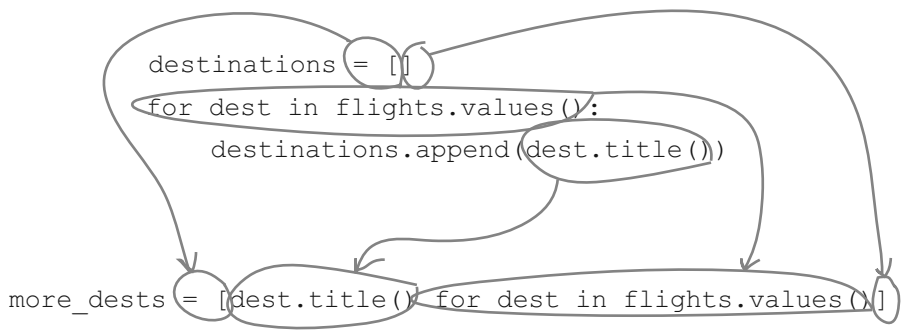
Теперь рассмотрим генератор более детально. Ниже снова приводятся исходные три строки кода и единственная строка с генератором, выполняющая ту же задачу.

Запомните: обе версии создают новые списки (`destinations` и `more_dests`) с теми же самыми данными.

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())
```

```
more_dests = [dest.title() for dest in flights.values()]
```

Выделим отдельные части трех исходных строк кода и посмотрим, где они используются в генераторе.



Обнаружив этот шаблон в другом коде, вы легко превратите его в генератор. Например, вот код, рассматривавшийся ранее (он производит список времени вылетов в 12-часовом формате), преобразованный в генератор:

```
flight_times = []
for ft in flights.keys():
    flight_times.append(convert2ampm(ft))
```

```
fts2 = [convert2ampm(ft) for ft in flights.keys()]
```

Этот код делает *то же самое*.

В чем соль?

Все эти генераторы кажутся малопонятными. Я бы использовала обычный цикл «for» для подобной работы. Думаете, стоит учиться правильно писать генераторы?

Да. Мы думаем, стоит.

Изучить работу генераторов стоит по двум причинам.

Во-первых, они требуют меньше кода (то есть дают меньше работы вашим пальцам) и интерпретатор Python оптимизирован для выполнения генераторов с максимальной скоростью. Это означает, что они выполняются *быстрее*, чем эквивалентный код с циклом `for`.

Во-вторых, генераторы можно использовать там, где нельзя циклы `for`. Фактически вы уже видели, что генераторы могут находиться *справа* от оператора присваивания, а обычные циклы `for` — нет. Это может очень пригодиться (как будет показано далее).

Генераторы используются не только со списками

Генераторы, которые вы видели до этого, создавали новые списки, поэтому их называют **генераторами списков**. Если генератор создает новый словарь, он называется **генератором словарей**. Чтобы ничего не упустить, добавим, что можно также определить **генератор множеств**.

Однако такого понятия, как *генератор кортежей*, не существует; причину мы объясним немного позже.

Для начала посмотрим, как работает генератор словарей.

Определение генератора словарей

Вспомним код в начале главы, читавший исходные данные из файла CSV в словарь `flights`. Потом эти данные преобразовывались в новый словарь `flights2`, в котором ключами являются времена вылета в 12-часовом формате, значениями — названия пунктов назначения в «регистре заголовка».

```
...

flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()

...
```

Этот код соответствует «шаблону генератора».

Преобразуем эти три строки кода в генератор словарей.

Сначала присвоим новый, пустой словарь переменной (с именем `more_flights`).

```
more_flights = {}
```

1. Создадим новый, пустой словарь.

Определим, как будут осуществляться итерации по существующим данным (в словаре `flights`) с помощью нотации `for` (не забудьте, что двоеточие в конце не нужно).

```
more_flights = {for k, v in flights.items()}
```

2. Итерации по каждой паре ключ/значение в существующих данных.

Обратите внимание: в конце НЕТ двоеточия.

Чтобы завершить генератор словарей, нужно определить, как должны получаться ключи и значения для нового словаря. Цикл `for` в начале страницы производит ключ, преобразуя время вылета в 12-часовой формат вызовом функции `convert2ampm`, а ассоциированное значение преобразуется в регистр заголовка вызовом метода `title`. Эквивалентный генератор должен выполнить то же самое и, так же как в случае с генератором списков, *слева* от ключевого слова `for`. Обратите внимание: новый ключ и новое значение разделены двоеточием.

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

3. Связать преобразованный ключ с преобразованным значением (обратите внимание на двоеточие).

Вот он — ваш первый генератор словарей. Давайте протестируем код, чтобы убедиться в его работоспособности.

Расширенные генераторы с фильтрами

Представьте, что нам потребовалось преобразовать только расписание рейсов до «Фрипорта».

Возвращаясь к начальному варианту цикла `for`, скорее всего, вы добавили бы выражение `if`, чтобы отфильтровать данные по текущему значению `v` (пункт назначения), как здесь.

```
just_freeport = {}
for k, v in flights.items():
    if v == 'FREEPORT':
        just_freeport[convert2ampm(k)] = v.title()
```

Преобразуется и добавляется в словарь «just_freeport» только расписание рейсов до пункта назначения «Фрипорт».

```
TIME,DESTINATION
09:35,FREEPORT
17:00,FREEPORT
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

Исходные данные.

Если выполнить этот код в командной строке >>>, он вернет всего две записи с данными (представляющие два запланированных вылета во «Фрипорт», которые содержатся в исходном файле). Здесь нет ничего удивительного — это стандартный способ использования инструкции `if` для фильтрации данных. Но, оказывается, такие фильтры можно использовать и в генераторах. Достаточно просто поместить инструкцию `if` (без двоеточия) в конец генератора. Вот генератор словарей, который уже встречался в конце предыдущей страницы.

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

А вот версия этого же генератора словарей, но уже с фильтром.

```
just_freeport2 = {convert2ampm(k): v.title() for k, v in flights.items() if v == 'FREEPORT'}
```

Преобразуется и добавляется в словарь «just_freeport2» только расписание рейсов до пункта назначения «Фрипорт».

Если выполнить этот генератор с фильтром в командной строке >>>, новый словарь `just_freeport2` будет заполнен теми же данными, что и `just_freeport`. Данные в `just_freeport` и `just_freeport2` являются **копией** данных из словаря `flights`.

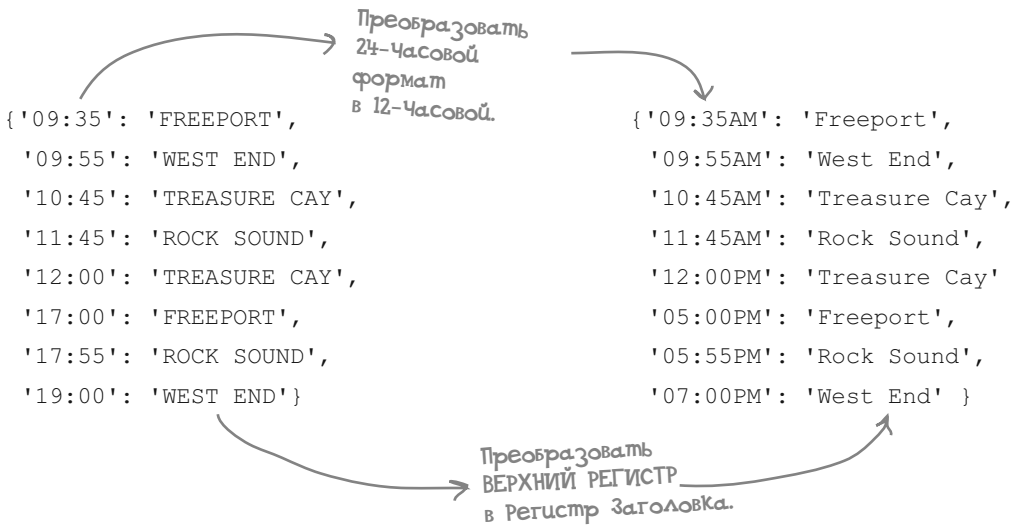
Правда, строка, создающая `just_freeport2`, кажется немного пугающей. Многие программисты, начинающие осваивать Python, жалуются, что генераторы **трудно читать**. Однако вспомните, что правило — конец строки означает конец инструкции — не применяется к коду между парой скобок, поэтому любой генератор можно переписать, разместив его на нескольких строках. Так код будет проще читаться.

```
just_freeport3 = {convert2ampm(k): v.title()
                  for k, v in flights.items()
                  if v == 'FREEPORT'}
```

Однострочный генератор можно сделать более простым для чтения. Кстати, программисты на Python все чаще записывают длинные генераторы на нескольких строках (поэтому такой синтаксис вы тоже будете встречать).

Вспомним, что нужно сделать

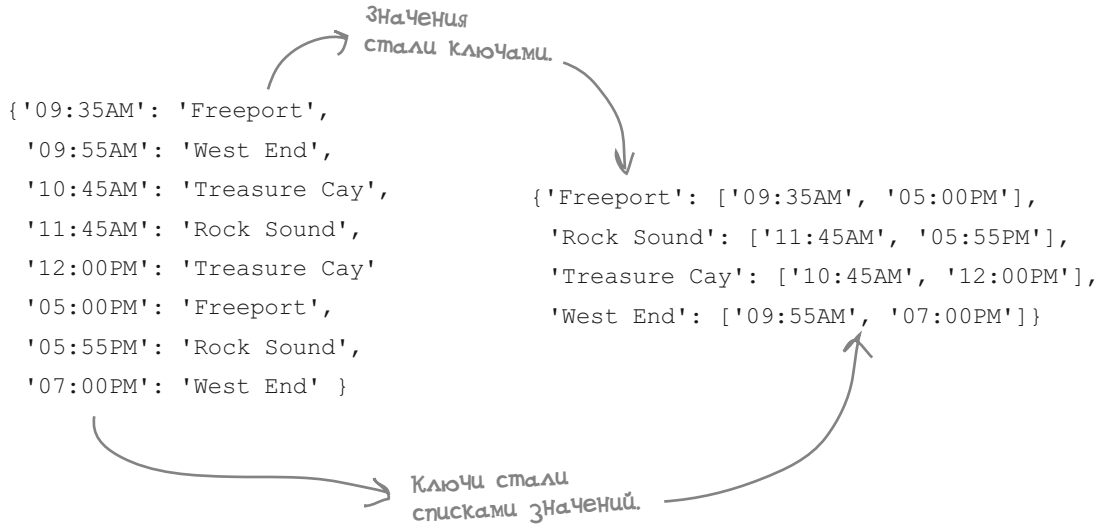
Теперь, ознакомившись с возможностями генераторов, вспомним, какие действия со словарями требовалось выполнить, и посмотрим, как их можно реализовать. Вот первое действие.



Используя данные из словаря `flights`, следующий генератор словарей выполняет необходимые преобразования *в одной строке кода*, заполняя новый словарь `fts` копиями данных.

```
fts = {convert2ampm(k): v.title() for k, v in flights.items()}
```

Второе действие (составить список с временем вылета для каждого пункта назначения) сложнее. И нам потребуется приложить больше усилий, потому что это более сложная операция.



Заточите карандаш

Пока мы не начали работу над второй операцией, проверим, как вы усвоили материал, связанный с генераторами.

Ваша задача — преобразовать три цикла `for` в генераторы. Решив ее, не забудьте протестировать свой код в IDLE (и только после этого перелистните страницу и прочитайте наше решение). Прежде чем попробовать написать генераторы, выполните эти циклы и посмотрите, что они делают. Напишите свое решение с генераторами в пропусках.

1

```
data = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
evens = []
for num in data:
    if not num % 2:
        evens.append(num)
```

Оператор% в Python выполняет деление по модулю, то есть возвращает остаток от деления первого числа на второе.

2

```
data = [ 1, 'one', 2, 'two', 3, 'three', 4, 'four' ]
words = []
for num in data:
    if isinstance(num, str):
        words.append(num)
```

Встроенная функция «isinstance» проверяет тип объекта, на который ссылается переменная.

3

```
data = list('So long and thanks for all the fish'.split())
title = []
for word in data:
    title.append(word.title())
```

Заточите карандаш Решение

Вам нужно было взять карандаш и изложить свои мысли на бумаге. Каждый из трех циклов `for` нужно было преобразовать в генератор, а затем протестировать свой код в IDLE.

1

```
data = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
evens = []
for num in data:
      if not num % 2:
            evens.append(num)
```

Эти четыре строки кода с циклом (который заполняет «evens») превратились в одну строку генератора.

```
evens = [ num for num in data if not num % 2 ]
```

2

```
data = [ 1, 'one', 2, 'two', 3, 'three', 4, 'four' ]
words = []
for num in data:
      if isinstance(num, str):
            words.append(num)
```

Снова четыре строки кода с циклом превратились в однострочный генератор.

```
words = [ num for num in data if isinstance(num, str) ]
```

3

```
data = list('So long and thanks for all the fish'.split())
title = []
for word in data:
      title.append(word.title())
```

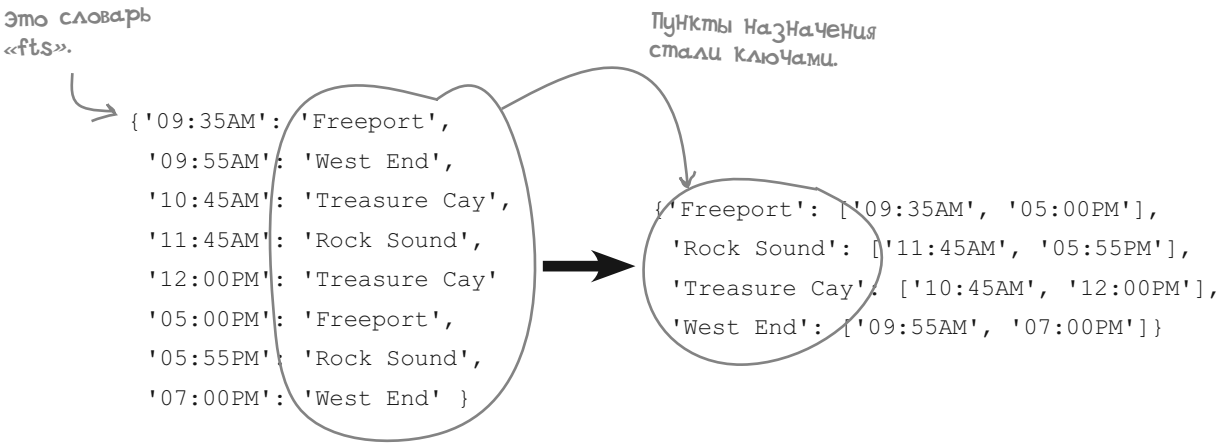
Этот пример самый простой из всех трех (потому что в нем нет фильтра).

```
title = [ word.title() for word in data ]
```

Преодолеваем сложности, как это принято в Python

После практики создания генераторов поэкспериментируем в командной строке >>> и выясним, что необходимо сделать с данными в словаре `fts`, чтобы преобразовать их в желаемое представление.

Прежде чем писать какой-либо код, рассмотрим необходимую трансформацию. Обратите внимание, что ключи в новом словаре (справа) — это список уникальных пунктов назначения, полученный из значений словаря `fts` (слева).



Оказывается, получить эти четыре уникальных пункта назначения очень просто. Так как данные хранятся в словаре `fts`, мы можем извлечь все значения с помощью `fts.values`, а затем передать их встроенной функции `set`, чтобы удалить повторения. Сохраним четыре уникальных названия пунктов назначения в переменной `dests`.

Получить все значения из «fts», затем передать их в аргументе встроенной функции «set». Это даст необходимый нам результат.

```
>>> dests = set(fts.values())
>>> print(dests)
{'Freeport', 'West End', 'Rock Sound', 'Treasure Cay'}
```

Четыре уникальных пункта назначения, которые можно использовать в качестве ключей в новом словаре.

Теперь, зная, как извлечь уникальные названия пунктов назначения, попробуем получить времена вылета, связанные с ними. Эти данные также хранятся в словаре `fts`.

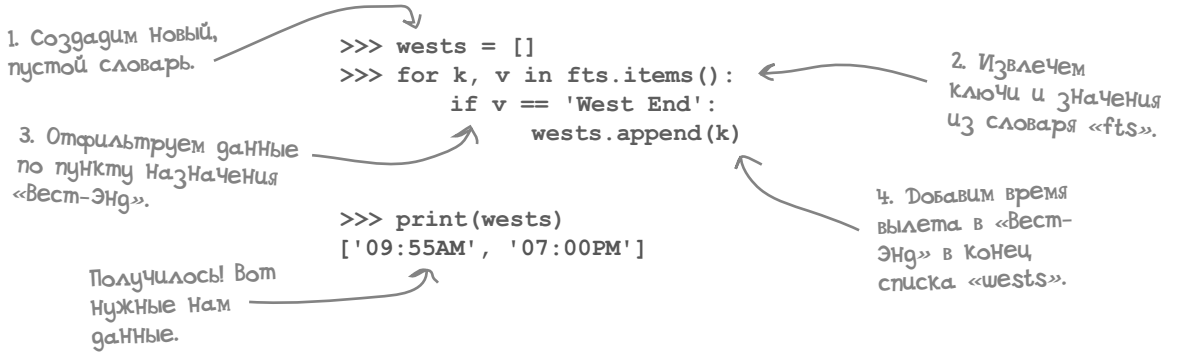
Пока вы не перевернули страницу, подумайте сами, как можно получить времена вылета для каждого из уникальных пунктов назначения.

Не стремитесь получить сразу все времена вылета для *каждого* пункта назначения; для начала попробуйте определить, как это сделать для Вест-Энда.

Выбираем время вылета для одного пункта назначения

Начнем с выборки времен вылета для одного пункта назначения, а именно Вест-Энда. Вот данные, которые нужно извлечь.

Как и раньше, откроем командную строку >>> и поэкспериментируем. Извлечь времена вылета в Вест-Энд из словаря `fts` можно с помощью этого кода.



Взглянув на этот код, вы, возможно, почувствовали внутри звон тревожного колокольчика, потому что этот цикл `for` наверняка можно преобразовать в генератор списков, да?

Циклу `for` выше соответствует такой генератор списков.

Четыре строки кода сократились до одной, благодаря использованию генератора списков.

```
>>> wests2 = [k for k, v in fts.items() if v == 'West End']

>>> print(wests2)
['09:55AM', '07:00PM']
```

Тоже получилось! Вот нужные нам данные.

Теперь, узнав, как извлечь данные для одного пункта назначения, проделаем то же самое для всех пунктов.

Получаем время вылета для всех пунктов назначения

Теперь у нас есть код, извлекающий множество уникальных названий пунктов назначения.

```
dests = set(fts.values())
```

Уникальные пункты назначения.

А еще у нас такой генератор списков, который извлекает список времени вылета для заданного пункта назначения (в этом примере для Вест-Энда).

```
wests2 = [k for k, v in fts.items() if v == 'West End']
```

Времена вылета в пункт назначения «Вест-Энд».

Чтобы получить список времен вылета для *всех* пунктов назначения, нужно объединить эти две инструкции (внутри цикла `for`).

В следующем коде мы избавились от переменных `dests` и `west2`, благодаря *непосредственному* использованию кода в цикле `for`. Теперь у нас нет жестко заданного названия *Вест-Энд*, потому что текущий пункт назначения хранится в `dest` (внутри генератора списков).

Уникальные пункты назначения.

```
>>> for dest in set(fts.values()):
        print(dest, '->', [k for k, v in fts.items() if v == dest])


Treasure Cay -> ['10:45AM', '12:00PM']
West End -> ['07:00PM', '09:55AM']
Rock Sound -> ['05:55PM', '11:45AM']
Freeport -> ['09:35AM', '05:00PM']
```

Времена вылетов в пункт назначения, название которого хранится в «dest».

Мы только что написали цикл `for`, который соответствует нашему шаблону создания генераторов, — и снова зазвонил маленький колокольчик в голове. Давайте попробуем пока не реагировать на этот звон, потому что код, который мы только что запустили в консоли >>>, *отображает* нужные нам данные... но на самом деле нам нужно сохранить данные в новом словаре. Создадим новый словарь (с именем `when`) и сохраним данные в нем. Вернемся в командную строку >>> и изменим тело цикла `for` так, чтобы данные сохранялись в словаре `when`.

1. Создадим новый, пустой словарь.

2. Извлечем множество уникальных названий пунктов назначения.

3. Добавим времена вылета в словарь «when».

```
>>> when = {}
>>> for dest in set(fts.values()):
        when[dest] = [k for k, v in fts.items() if v == dest]


>>> pprint.pprint(when)
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['05:55PM', '11:45AM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['07:00PM', '09:55AM']}
```

Вот и все: нужные данные теперь хранятся в словаре «when».

Если вы похожи на нас, то маленький колокольчик в вашей голове (который вы пытались заглушить) просто разрывается от звона и сводит вас с ума от одного только взгляда на этот код.

Чувство, возникающее, когда...

...единственная строка кода начинает походить на *заклинание*.

Отключите колокольчик в своей голове и снова посмотрите на код последнего цикла `for`.

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```

Он соответствует шаблону, который дает потенциальную возможность превратить его в генератор. Вот результат преобразования цикла `for` в генератор словарей, который извлекает *копии* нужных нам данных в новый словарь `when2`.

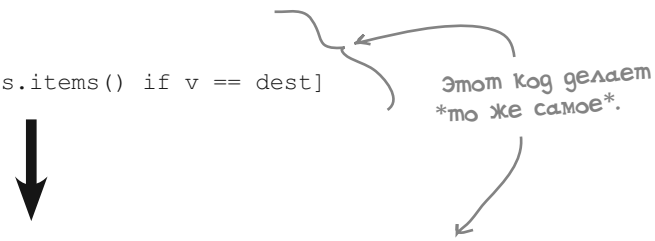
```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

Выглядит довольно *загадочно*, не так ли?

Это самый сложный генератор, который вы видели до сих пор, потому что *внешний* генератор словарей содержит *внутренний* генератор списков. Кстати, этот генератор словарей демонстрирует одну из особенностей, отличающих генераторы от эквивалентных циклов `for`: генератор можно поместить почти в любой части кода. Подобное невозможно с циклом `for`, который может быть только отдельной инструкцией (а не частью выражения).

Конечно, нет необходимости говорить, что *всегда* следует делать нечто подобное.

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```




```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

Будьте внимательны: генератор словарей, содержащий встроенный генератор списков, труднее читать, *когда вы видите его впервые*.

Однако по мере обретения опыта генераторы становятся проще для чтения и понимания, и — как уже говорилось в начале этой главы — программисты на Python *часто* используют их. Использовать генераторы или нет — целиком дело вкуса. Если вас больше привлекает цикл `for`, пользуйтесь им на здоровье. Если вам нравятся генераторы, используйте их... но не считайте себя *обязанными* это делать.

Пробная поездка

Прежде чем двигаться дальше, поместим весь код с генераторами в отдельный файл `do_convert.py`. Мы можем затем запустить этот код (в IDLE), чтобы убедиться, что все преобразования соответствуют требованиям *Bahamas Buzzers*. Проверьте, что ваш код соответствует нашему, затем запустите его, чтобы убедиться, что все работает как заявлено.

do_convert.py - /Users/paul/Desktop/_NewBook/ch12/do_convert.py (3.5.2)

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

Ln: 1 Col: 0

Python 3.5.2 Shell

```
/ch12/do_convert.py ==========
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}

{'05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End',
 '09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'}

{'Freeport': ['05:00PM', '09:35AM'],
 'Rock Sound': ['05:55PM', '11:45AM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['07:00PM', '09:55AM']}

>>>
```

Ln: 214 Col: 4

1. Исходные данные из файла CVS. Это словарь «flights».

2. Исходные данные, скопированные и преобразованные в 12-часовой формат и регистр заголовка. Это словарь «fts».

3. Расписание вылета в каждый из пунктов назначения (извлеченный из «fts»). Это словарь «when».

это не Глупые вопросы

В: Итак… сразу к делу: генератор — это только краткий синтаксис записи стандартной конструкции цикла?

О: Да, специально для цикла `for`. Стандартный цикл `for` и эквивалентный ему генератор выполняют одну и ту же работу. Однако генераторы выполняются значительно быстрее.

В: Как узнать, когда использовать генераторы списков?

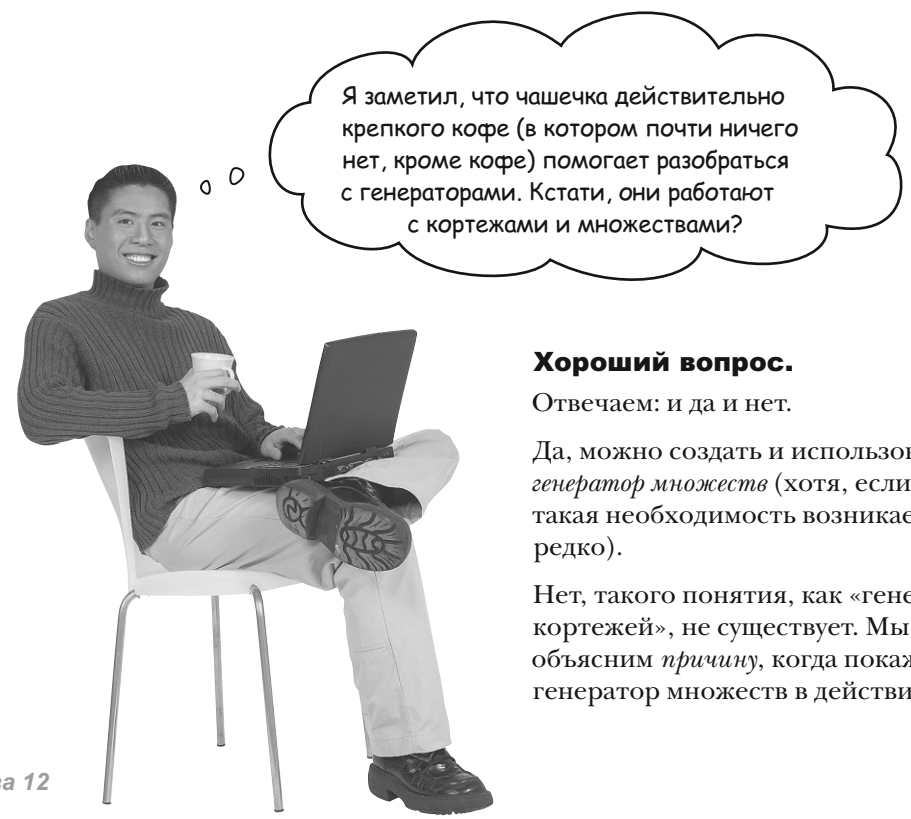
О: Здесь нет особого рецепта. Обычно если вы создаете новый список из существующего, хорошенько взгляните на код цикла. И спросите себя: можно ли преобразовать этот цикл в генератор. Если новый список — «временный» (то есть используется один раз и затем выбрасывается), ответьте на вопрос: подходит ли он на роль *встроенного* генератора списков. Основное правило — избегайте создания временных переменных в коде, особенно если они используются один раз. Спросите себя: можно ли в этом случае использовать генератор.

В: Можно ли вообще обойтись без генераторов?

О: Да, можно. Однако они широко используются в сообществе Python, и если вы не собираетесь обойтись без изучения чужого кода, мы советуем потратить время на знакомство с технологией генераторов в Python. Привыкнув к ним, вы уже не сможете без них обходиться. Кстати, мы говорили, что они работают быстрее?

В: Да, это понятно, но разве скорость выполнения имеет какое-то значение в наши дни? У меня суперскоростной компьютер, и он достаточно быстро выполняет циклы `for`.

О: Это интересное наблюдение. Конечно, теперь у нас есть компьютеры, которые значительно мощнее, чем были когда-либо. И мы тратим меньше времени, пытаясь оптимизировать каждый цикл CPU в своем коде (потому что, честно говоря, нам вообще не нужно это делать). Однако если есть технология, позволяющая повысить производительность, то почему бы ею не воспользоваться? Порой небольшие усилия дают существенный прирост производительности.



Хороший вопрос.

Отвечаем: и да и нет.

Да, можно создать и использовать *генератор множеств* (хотя, если честно, такая необходимость возникает редко).

Нет, такого понятия, как «генератор кортежей», не существует. Мы объясним *причину*, когда покажем генератор множеств в действии.

Генератор множеств в действии

Генераторы множеств позволяют создавать новые множества одной строкой кода, используя конструкцию, очень похожую на генератор списков.

Что отличает генератор множеств от генератора списков, так это то, что генераторы множеств заключены в фигурные скобки (а не в квадратные, как генераторы списков). Генераторы словарей тоже заключены в фигурные скобки, и это немного путает. (Иногда непонятно, что двигало разработчиками Python, когда они это придумали.)

Литерал множества заключается в фигурные скобки, как и литерал словаря. Чтобы отличать одно от другого, обратите внимание на двоеточие, которое служит символом-разделителем в словарях, но не используется в множествах. Подобный совет применим также для различения генераторов словарей и множеств: смотрите на двоеточие. Если оно присутствует — это генератор словарей. Если нет — генератор множеств.

Вот небольшой пример генератора множеств (который имеет отношение к другому примеру из книги). Дано множество букв (`vowels`) и строка (`message`); цикл `for` и эквивалентный ему генератор множеств производят один и тот же результат — множество гласных, найденных в `message`.

```
vowels = {'a', 'e', 'i', 'o', 'u'}
message = "Don't forget to pack your towel."

found = set()
for v in vowels:
    if v in message:
        found.add(v)
```

Генератор множеств соответствует тому же шаблону, что и генератор списков.

```
found2 = { v for v in vowels if v in message  }
```

Обратите внимание: здесь используются фигурные скобки, потому что этот генератор создает множество.

Поэкспериментируйте с кодом на этой странице в командной строке >>>. Вы уже знаете, что могут делать генераторы списков и словарей, поэтому понять работу генераторов множеств вам будет совсем не сложно. В них нет ничего особенного, кроме того, о чем говорилось на этой странице.

Как распознать генератор в коде

Чем чаще вы будете видеть генераторы в коде, тем проще вам будет узнать их и понять, как они работают. Вот хорошее общее правило, как узнать генератор списка:

Если вы видите код внутри [и], значит перед вами генератор списков.

Это правило можно обобщить.

Если вы видите код внутри скобок (фигурных или квадратных), скорее всего, перед вами генератор.

Почему «скорее всего»?

Кроме квадратных скобок `[]`, генераторы могут заключаться, как вы уже видели, в фигурные скобки `{}`. Если код заключен между `[и]`, это генератор **списков**. Если код заключен между `{и}`, это генератор **множеств** или **словарей**. Генератор словарей легко отличить по использованию двоеточия в качестве разделителя.

Однако код может быть заключен между `(и)`, что является *особым случаем*, и вам можно простить, если вы посчитали, что код внутри круглых скобок должен быть непременно *генератором кортежей*. Вас можно простить, но вы ошибаетесь: «генераторов кортежей» не существует, даже притом что можно поместить код между `(и)`. После того «веселья», которое повстречалось вам при изучении генераторов в этой главе, вы, наверное, думаете: *что может быть еще более странным?*

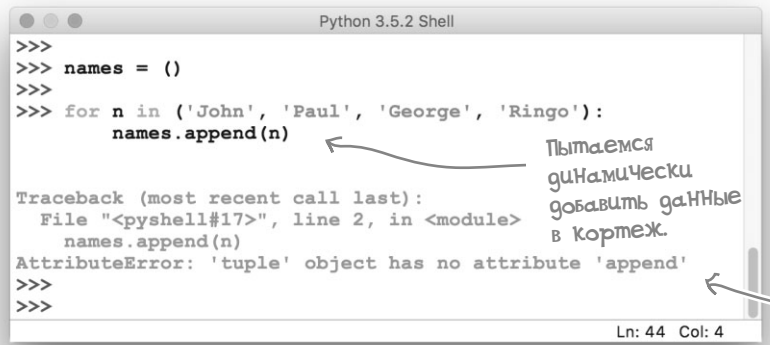
В завершение главы (и книги) посмотрим, что произойдет, если поместить код между `(` и `)`. Это не «генератор кортежей», но такой код допустим. Тогда что это?

А что с «генераторами кортежей»?

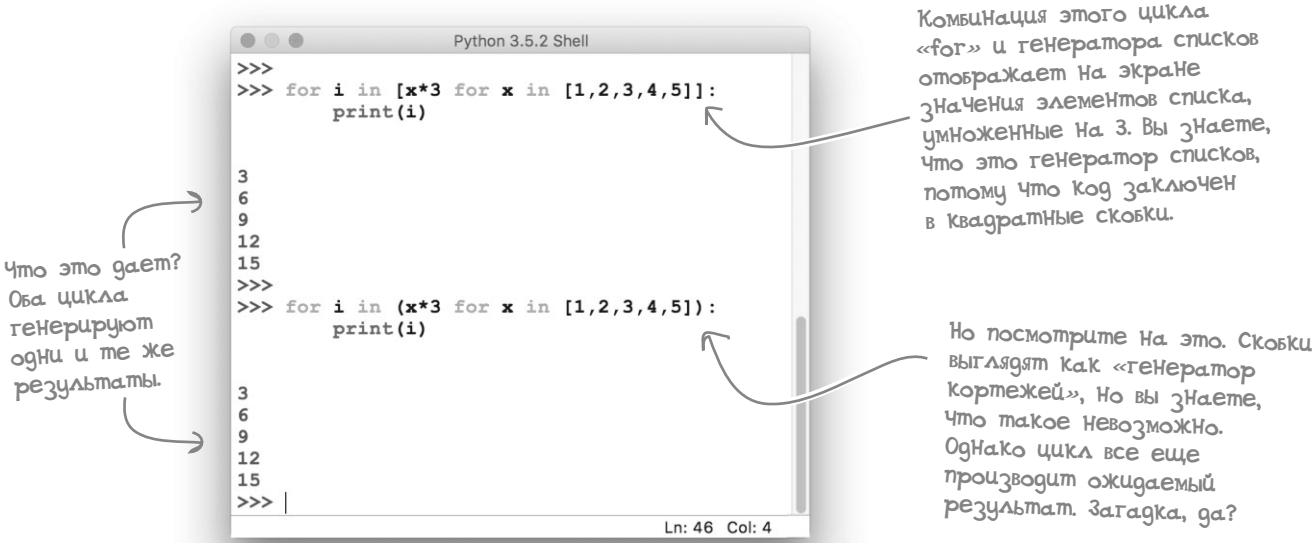
Четырем встроенным структурам данных в Python (кортежи, списки, множества и словари) можно найти массу применений. Однако они все, кроме кортежей, могут создаваться с помощью генераторов.

Почему так?

Оказывается, идея «генераторов кортежей» не имеет особого смысла. Как вы помните, кортежи — *неизменяемые* структуры: созданные однажды, кортежи не могут изменяться. Это также означает, что в коде нельзя изменять значения кортежей, как демонстрирует следующий сеанс IDLE.



Здесь нет ничего странного или удивительного, потому что такое поведение вполне ожидаемо: если кортеж существует, его *нельзя* изменить. Этого одного факта достаточно, чтобы исключить использование кортежей в каких-либо генераторах. Но взгляните на следующий сеанс в командной строке >>>. Второй цикл отличается от первого совсем немного: квадратные скобки вокруг генератора списков (в первом цикле) были заменены круглыми (во втором).



Круглые скобки вокруг кода == Генератор

Когда вы видите что-то вроде генератора списков, но в круглых скобках, перед вами **выражение-генератор**:

Это похоже на генератор списков, но им не является: это выражение-генератор.

```
for i in (x*3 for x in [1, 2, 3, 4, 5]):
    print(i)
```

Выражение-генератор можно использовать везде, где используется генератор списков, оно дает такой же результат.

Как вы уже видели в конце предыдущей страницы, если квадратные скобки в генераторе списков заменить круглыми, результат не изменится, потому что выражения-генераторы и генераторы списков дают один и тот же результат.

Однако они выполняются по-разному.

Если вы глубоко задумались над значением предыдущего предложения, примите во внимание следующее: когда выполняется генератор списков, **все** данные создаются, прежде чем их обработка продолжится. Если говорить в контексте примера, показанного в начале страницы, цикл `for` не начнет обработку данных, созданных генератором списков, пока выполняется сам генератор. То есть если для создания данных потребуется много времени, это отсрочит выполнение любого другого кода, *пока не выполнится сам генератор списков*.

Если списки небольшие (как в примере), это не проблема.

Но представьте, что генератору требуется создать список, содержащий порядка 10 миллионов элементов. Теперь у вас две проблемы: (1) придется подождать, пока генератор списков обработает эти 10 миллионов элементов, *прежде чем сделать что-то еще*; (2) у вашего компьютера может не хватить оперативной памяти, чтобы вместить все данные, созданные генератором списков (**10 миллионов** различных элементов данных). Если генератор списков израсходует всю память, интерпретатор завершится с ошибкой (и программа тоже).

Генераторы списков и выражения-генераторы дают один и тот же результат, но выполняются по-разному.

Выражения-генераторы производят элементы данных по одному...

Если заменить квадратные скобки в генераторе списков круглыми, генератор списков превратится в **выражение-генератор** и поведение программы изменится.

В отличие от генераторов списков, которые должны завершить работу прежде, чем выполнится любой другой код, выражение-генератор возвращает данные по мере их создания. Если вы генерируете 10 миллионов элементов данных, интерпретатору потребуется память только для хранения **одного** из них, а код, ожидающий данных от генератора, выполняется сразу же — *без ожидания*.

Пример поможет лучше понять разницу при использовании выражения-генератора, поэтому решим простую задачу дважды: один раз с использованием генератора списков, а затем с помощью выражения-генератора.

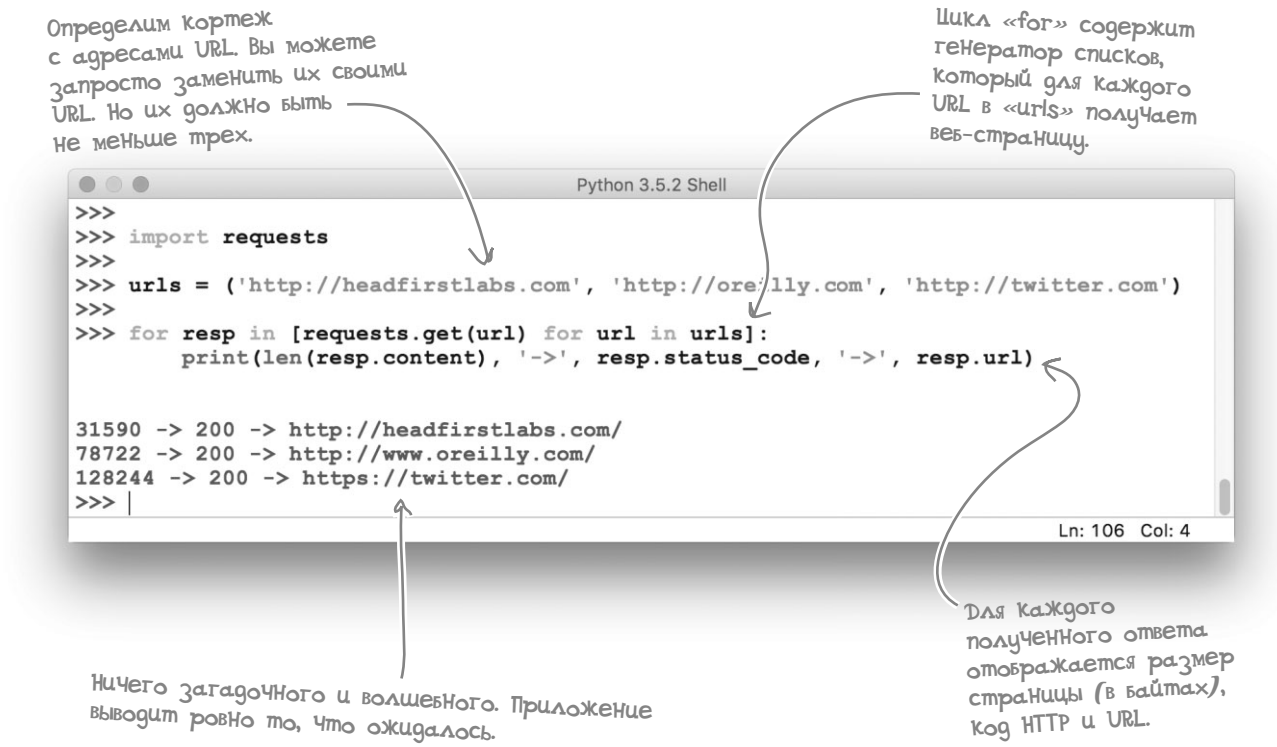
Использование генератора списков для обработки URL

Чтобы продемонстрировать отличия в применении выражения-генератора, решим задачу с использованием генератора списков (прежде чем преобразовать его в выражение-генератор).

Как принято в этой книге, поэкспериментируем в консоли >>>, воспользовавшись библиотекой `requests` (которая позволяет программно взаимодействовать с веб). В следующем коротком сеансе мы импортировали библиотеку `requests`, определили кортеж с тремя элементами (с именем `urls`) и с помощью цикла `for` и генератора списков запросили страницы для каждого URL, а потом обработали полученные ответы.

Загрузите «requests» из каталога PyPI командой «pip».

Чтобы понять происходящее, точно следуйте за нами.



Если вы выполняли этот пример на своем компьютере, то наверняка заметили значительную задержку между вводом цикла `for` и получением результатов. Когда результаты готовы, они отображаются за один прием (все сразу). Это происходит потому, что генератор списков обрабатывает каждый из адресов URL в `urls`, прежде чем результаты станут доступны циклу `for`. Что в итоге? Вам приходится ждать вывода на экран.

Обратите внимание, что в этом коде *все верно*: он делает то, что от него требуется, вывод получается правильным. Но давайте преобразуем генератор списков в выражение-генератор, чтобы увидеть разницу. Как упоминалось выше, повторяйте за нами на своем компьютере примеры, которые приводятся на следующей странице (и вы увидите, что произойдет).

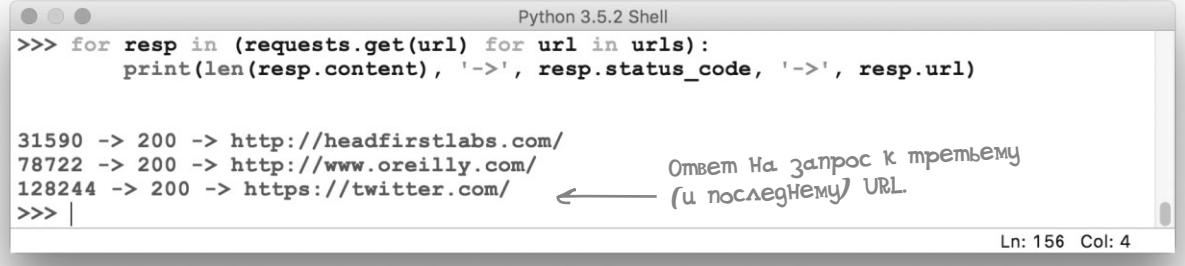
Использование выражения-генератора для обработки URL

Вот пример с предыдущей страницы, переделанный для использования выражения-генератора. Сделать это очень легко; просто замените квадратные скобки в генераторе списков круглыми.

Через небольшой интервал времени после входа в цикл `for` появляется первый результат.

Затем, мгновение спустя, появляется следующая строка с результатом.

Наконец, через несколько мгновений появляется строка с последним результатом (и цикл `for` завершает свою работу).

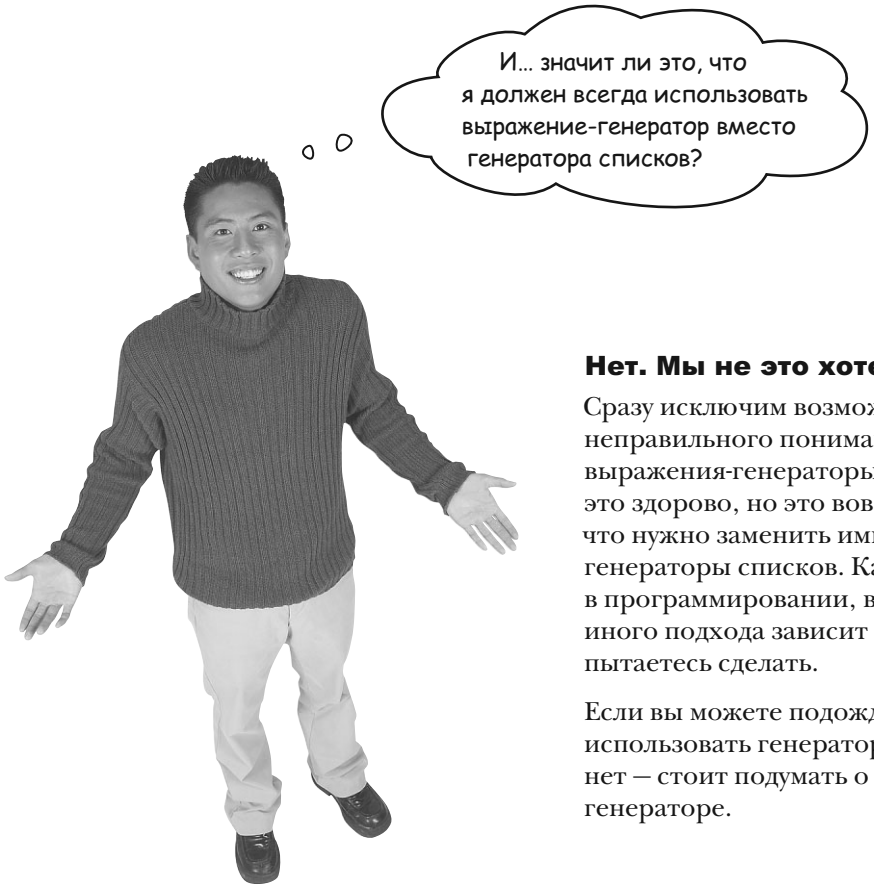


Использование выражения-генератора: что произошло?

Если сравнить результаты, полученные с помощью генератора списков и при использовании выражения-генератора, можно заметить, что они *идентичны*. Но поведение кода отличается.

Генератор списков **ожидает** получения всех данных, а затем возвращает их циклу `for`, а выражение-генератор **выдает** данные по мере их поступления. Это означает, что цикл `for`, использующий выражение-генератор, выглядит более отзывчивым, чем тот же цикл, но использующий генератор списков (который заставляет вас ждать).

Если вы думаете, что на самом деле это не так важно, представьте, что кортеж содержит сотню, тысячу или миллион адресов URL. А теперь вообразите, что процесс, обрабатывающий ответы, поставляет данные другому процессу (например, базе данных). С увеличением числа URL поведение генератора списков все сильнее будет отличаться от поведения выражения-генератора.



Нет. Мы не это хотели сказать.

Сразу исключим возможность неправильного понимания: выражения-генераторы существуют — это здорово, но это вовсе не означает, что нужно заменить ими все генераторы списков. Как и многое в программировании, выбор того или иного подхода зависит от того, что вы пытаетесь сделать.

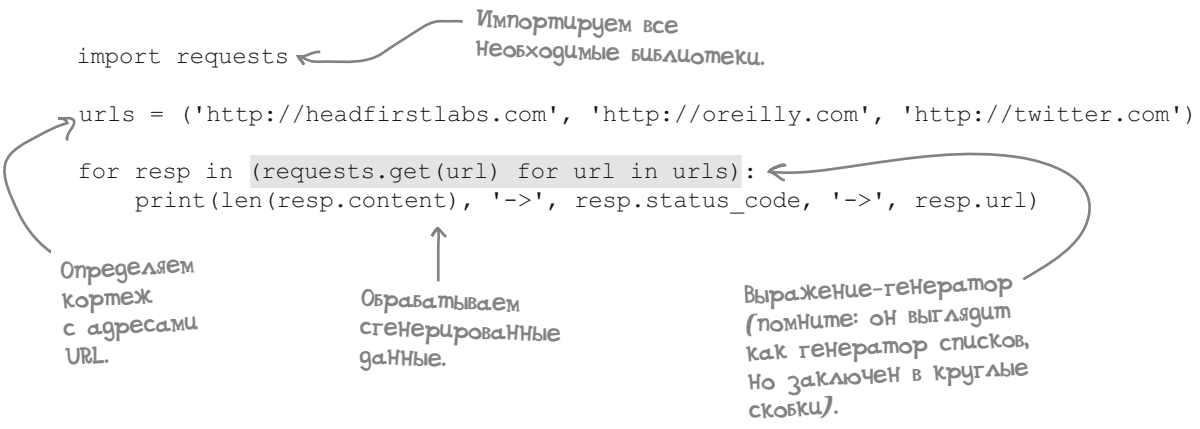
Если вы можете подождать — можно использовать генератор списков; если нет — стоит подумать о выражении-генераторе.

Одна из интересных возможностей для использования выражения-генератора — встраивание в функцию. Подумаем, как наш только что созданный генератор можно встроить внутрь функции.

Определим, что должна делать функция

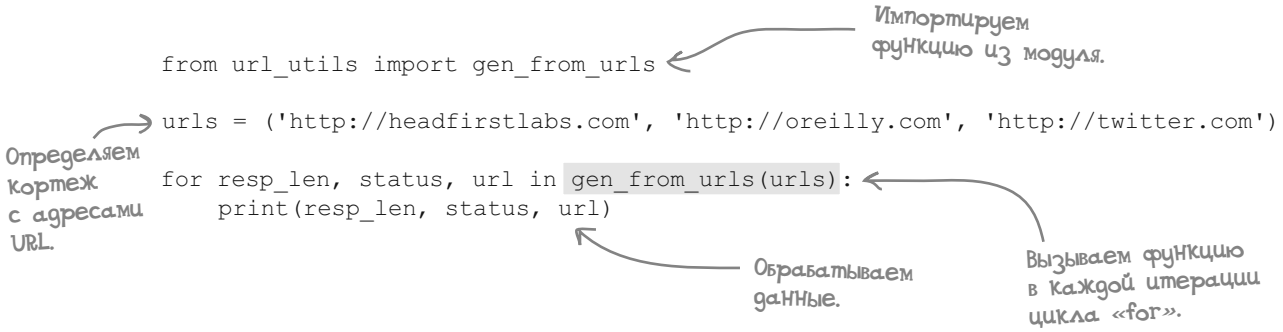
Представьте, что вы хотите поместить выражение-генератор `requests` внутрь функции. Вы решили упаковать генератор в маленький модуль и хотите, чтобы другие программисты могли использовать его, не зная и не понимая выражений-генераторов.

Вот как выглядит код выражения-генератора.



Создадим функцию, которая инкапсулировала бы этот код. Функция, которую мы назовем `gen_from_urls`, принимает один аргумент (кортеж URL) и возвращает кортеж результатов для каждого URL. Возвращаемый кортеж содержит три значения: размер страницы, код HTTP и URL, откуда пришел ответ.

Предположим, что функция `gen_from_urls` уже существует и вы хотите, чтобы другие программисты могли вызывать ее в заголовке цикла `for`, как показано ниже.



Этот код мало отличается от кода в начале страницы, но обратите внимание, что программисты, использующие `gen_from_urls`, не знают (и не должны знать) об использовании `requests` для отправки веб-запросов. И еще им нет необходимости знать, что вы используете выражение-генератор. Все детали реализации скрыты за простым и понятным фасадом вызова функции.

Посмотрим, как можно написать `gen_from_urls`, чтобы она генерировала нужные нам данные.

Ощути силу функций-генераторов

Теперь, узнав, что должна делать функция `gen_from_urls`, напишем ее. Начнем с создания нового файла `urls_utils.py`. Откройте этот файл в редакторе, затем добавьте `import requests` в его начало.

Строка `def` объявления функции выглядит просто — она принимает один кортеж и возвращает другой (обратите внимание, как мы добавили аннотации типов, чтобы пользователям было все понятно). Сделаем шаг вперед и добавим строку `def` в файл, примерно так.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
```

После импорта «requests» определим новую функцию.

Тело функции — генератор с предыдущей страницы, строка `for` получена простым копированием-вставкой.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
```

Добавьте генератор в заголовок цикла «for».

Следующая строка кода должна «вернуть» результат GET-запроса, выполненного функцией `requests.get`. Несмотря на искушение добавить такую строку в тело цикла `for`, **пожалуйста, не делайте этого.**

```
return len(resp.content), resp.status_code, resp.url
```

Выполнив инструкцию `return`, функция *завершится*, но это не то, что нам нужно, потому что функция `gen_from_urls` вызывается в заголовке цикла `for`, который ожидает получить *следующий* кортеж при *каждом вызове функции*.

Но если нельзя выполнить `return`, что же делать?

Используйте `yield`. Ключевое слово `yield` добавлено в Python для поддержки **функций-генераторов**, и его можно использовать везде, где используется `return`. Когда выполняется такая замена, функция превращается в функцию-генератор, и ее можно «вызывать» из любого итератора, в нашем случае это цикл `for`.

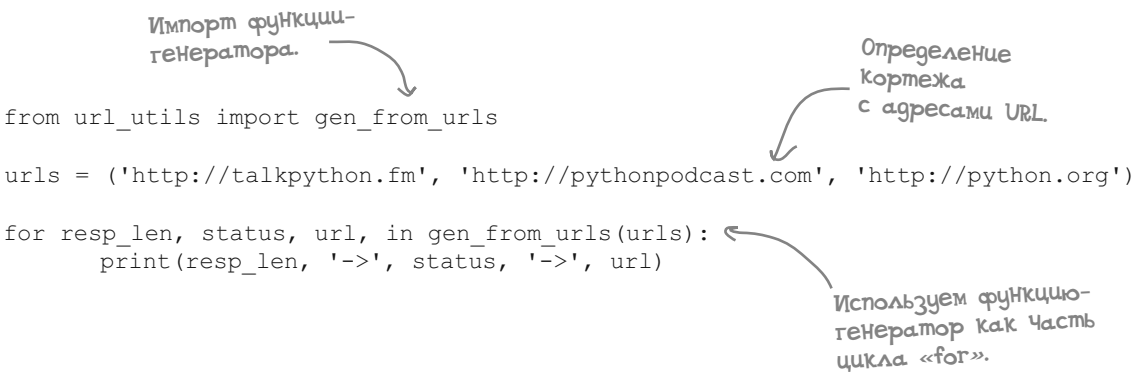
Используйте «yield», чтобы вернуть циклу «for» результат каждого отдельного запроса GET. Помните: НЕ используйте «return».

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Посмотрим внимательнее, что здесь происходит.

Исследуем функцию-генератор, 1 из 2

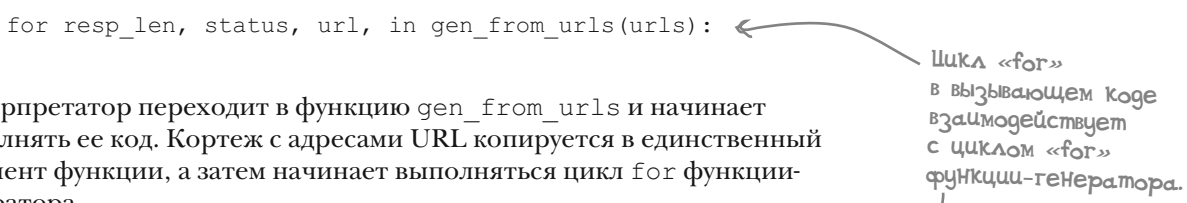
Чтобы понять, что происходит во время работы функции-генератора, исследуем работу следующего кода.




```
from url_utils import gen_from_urls

urls = ('http://talkpython.fm', 'http://pythonpodcast.com', 'http://python.org')

for resp_len, status, url, in gen_from_urls(urls):
       print(resp_len, '->', status, '->', url)
```

Первые две строки просты и понятны: они импортируют функцию и создают кортеж с адресами URL.

Самое интересное начинается в следующей строке, где вызывается функция-генератор `gen_from_urls`. Будем называть этот цикл `for` «вызывающим кодом».




```
for resp_len, status, url, in gen_from_urls(urls):
```

Интерпретатор переходит в функцию `gen_from_urls` и начинает выполнять ее код. Кортеж с адресами URL копируется в единственный аргумент функции, а затем начинает выполняться цикл `for` функции-генератора.

```
def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Цикл `for` содержит выражение-генератор, которое извлекает первый URL из кортежа `urls` и посылает запрос GET по указанному адресу. Когда приходит HTTP-ответ, выполняется инструкция `yield`.

Здесь все становится еще интереснее (или загадочнее — это уж как посмотреть).

Вместо перехода к следующему URL в кортеже `urls` (то есть к следующей итерации в цикле `for` внутри `gen_from_urls`) `yield` передает три элемента данных вызывающему коду. Вместо завершения функция-генератор `gen_from_urls` *приостанавливается*, как будто *нажали клавишу «Пауза»*...

Исследуем функцию-генератор, 2 из 2

Когда данные (переданные инструкцией `yield`) поступают в вызывающий код, выполняется тело цикла `for`. Тело содержит единственный вызов функции `print`. Он выполняется, и на экране появляются результаты обработки первого URL.

```
print(resp_len, '->', status, '->', url)
```

```
34591 -> 200 -> https://talkpython.fm/
```

Затем цикл `for` в вызывающем коде начинает новую итерацию, вызывая `gen_from_urls` снова... вроде бы.

Вроде бы, но *не совсем*. На самом деле `gen_from_urls` возобновляет работу с того места, где она была приостановлена. Цикл `for` внутри `gen_from_urls` начинает новую итерацию, извлекает следующий URL из кортежа `urls`, посылает запрос серверу, указанному в этом URL. Когда приходит HTTP-ответ, выполняется инструкция `yield`, возвращающая три элемента данных вызывающему коду (эти данные функция получает из объекта `resp`).

Три необходимых элемента данных функция получает из объекта «resp», который возвращает метод «get» из библиотеки «requests».

```
yield len(resp.content), resp.status_code, resp.url
```

Как и в прошлый раз, вместо завершения функция-генератор `gen_from_urls` снова *приостанавливается*, как будто *нажали клавишу «Пауза»*...

Когда данные (переданные инструкцией `yield`) поступают в вызывающий код, снова выполняется функция `print` в теле цикла `for`, выводя на экран второй результат.

```
34591 -> 200 -> https://talkpython.fm/
19468 -> 200 -> http://pythonpodcast.com/
```

Далее цикл `for` в вызывающем коде начинает следующую итерацию и вновь «вызывает» функцию `gen_from_urls`, возобновляя ее выполнение. Функция выполняет инструкцию `yield`, результаты возвращаются вызывающему коду и снова выводятся на экран.

```
34591 -> 200 -> https://talkpython.fm/
19468 -> 200 -> http://pythonpodcast.com/
47413 -> 200 -> https://www.python.org/
```

После обработки последнего URL в кортеже циклы `for` в функции-генераторе и вызывающем коде завершают свою работу. Выглядит так, будто два фрагмента кода работали поочередно, передавая данные друг другу.

Посмотрим, как это работает в консоли >>>. Пришло время для последней «Пробной поездки».

Пробная поездка

В этой последней «Пробной поездке» протестируем работу функции-генератора. Как обычно, введите код в окне редактора IDLE и нажмите F5, чтобы начать экспортировать функцию в консоль >>>. Следуйте за нами.

Вот генераторная функция «gen_from_urls» в модуле «url_utils.py».

url_utils.py - /Users/paul/Desktop/_NewBook/ch12/url_utils.py (3.5.2)

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Ln: 8 Col: 0

Первый пример демонстрирует вызов функции `gen_from_urls` в заголовке цикла `for`. Вы должны получить примерно тот же результат, что и несколькими страницами ранее.

Второй пример демонстрирует вызов `gen_from_urls` в генераторе словарей. Обратите внимание, что в новом словаре сохраняются только URL (как ключ) и размер загруженной страницы (как значение). Код состояния HTTP не используется в этом примере, поэтому мы сообщили интерпретатору, что его можно игнорировать, использовав **имя переменной по умолчанию** (одиночный символ подчеркивания).

Каждый результат появляется после небольшой паузы, по мере получения данных внутри функции.

Генератор словарей связывает URL с размером загруженной страницы.

Передаем кортеж с адресами URL в функцию-генератор.

Символ подчеркивания сообщает, что код состояния HTTP игнорируется.

Вывод содержимого словаря «url_res» в отформатированном виде доказывает, что функцию-генератор можно использовать внутри генератора словарей (как и внутри цикла «for»).

Python 3.5.2 Shell

```
>>>
>>>
>>> for resp_len, status, url in gen_from_urls(urls):
        print(resp_len, '->', status, '->', url)


31590 -> 200 -> http://headfirstlabs.com/
78722 -> 200 -> http://www.oreilly.com/
128244 -> 200 -> https://twitter.com/
>>>
>>> urls_res = {url: size for size, _, url in gen_from_urls(urls)}
>>>
>>> import pprint
>>>
>>> pprint.pprint(urls_res)
{'http://headfirstlabs.com/': 31590,
 'http://www.oreilly.com/': 78722,
 'https://twitter.com/': 128244}
>>>
>>>
```

Ln: 271 Col: 0

Заключительные заметки

Использование генераторов считается сложной темой в мире Python. Однако во многом это связано с отсутствием похожих возможностей в других распространенных языках программирования, поэтому программистам, перешедшим с них на Python, они зачастую даются нелегко (потому что они раньше с ними не сталкивались).

Но программисты на Python из *Head First Labs любят* генераторы и убеждены, что при многократном повторении их использование в конструкциях циклов станет второй натурой. Они не могут представить, как можно обходиться без генераторов.

Даже если вам кажется, что синтаксис генераторов сложен для понимания, постарайтесь привыкнуть к нему. Если вы не считаете, что генераторы имеют лучшую производительность по сравнению с эквивалентными циклами `for`, возможность использовать генераторы там, где нельзя использовать `for`, — достаточная причина, чтобы отнестись со всей серьезностью к этому механизму в языке Python. Когда вы привыкнете к их синтаксису, возможности для использования генераторов станут проявляться сами собой, их подскажет внутренний голос, как в случаях с применением функций здесь, циклов там, классов где-то еще и т. д. Далее кратко перечислено, с чем вы ознакомились в этой главе.

КОНТРОЛЬНЫЙ СПИСОК

- Для работы с данными в файлах в Python имеется несколько возможностей. Кроме стандартной функции `open`, для обработки данных в формате CSV можно воспользоваться средствами из модуля CSV, который входит в стандартную библиотеку.
- **Цепочки** вызовов методов позволяют произвести обработку данных в одной строке кода. Цепочка `string.strip().split()` часто встречается в коде на Python.
- Обращайте внимание на порядок следования методов в цепочке. Особенно на тип данных, возвращаемых каждым из них (типы должны быть совместимы с вызываемыми методами).
- Цикл `for`, используемый для преобразования данных из одного формата в другой, можно представить в виде **генератора списка, словаря или множества**.
- Генераторы можно использовать для обработки существующих списков, словарей и множеств, но чаще других «в дикой природе» встречаются генераторы списков.
- **Генераторы списков** — это код, заключенный в квадратные скобки, а **генераторы словарей** — код в фигурных скобках (с двоеточием в качестве разделителя). **Генераторы множеств** также заключены в фигурные скобки (но без двоеточия-разделителя).
- Не существует такого понятия, как «генератор кортежей», потому что кортежи — неизменяемые структуры (поэтому нет смысла пытаться изменять их динамически).
- Если вы увидели в коде генератор, заключенный в круглые скобки, перед вами **выражение-генератор** (которое можно превратить в функцию, использующую ключевое слово `yield` для возврата нужных данных).

Поскольку эта глава завершается (и, по определению, завершается вся книга), у нас остался последний вопрос к вам. Глубоко вдохните и переверните страницу.

Последний вопрос

ОК. Вот наш последний вопрос: *дойдя до этой страницы, вы заметили, что в программах на Python много пробелов?*

Наиболее часто от начинающих программистов Python приходится слышать жалобы, что в языке используется слишком много пробелов для выделения блоков кода (вместо того чтобы, например, использовать фигурные скобки). Но через некоторое время вы привыкнете и не будете ничего замечать.

Это не случайно: создатель языка преднамеренно решил использовать в Python много пробелов.

Язык специально разработан так, потому что **код читают чаще, чем пишут**. Код, имеющий непротиворечивое и хорошо известное оформление, легче читается. Благодаря использованию пробелов, код на Python, написанный 10 лет назад совершенно незнакомым вам человеком, по-прежнему легко читается *сегодня*.

И это огромное достижение всего сообщества Python, а значит, и *ваше*.

Код из главы 12

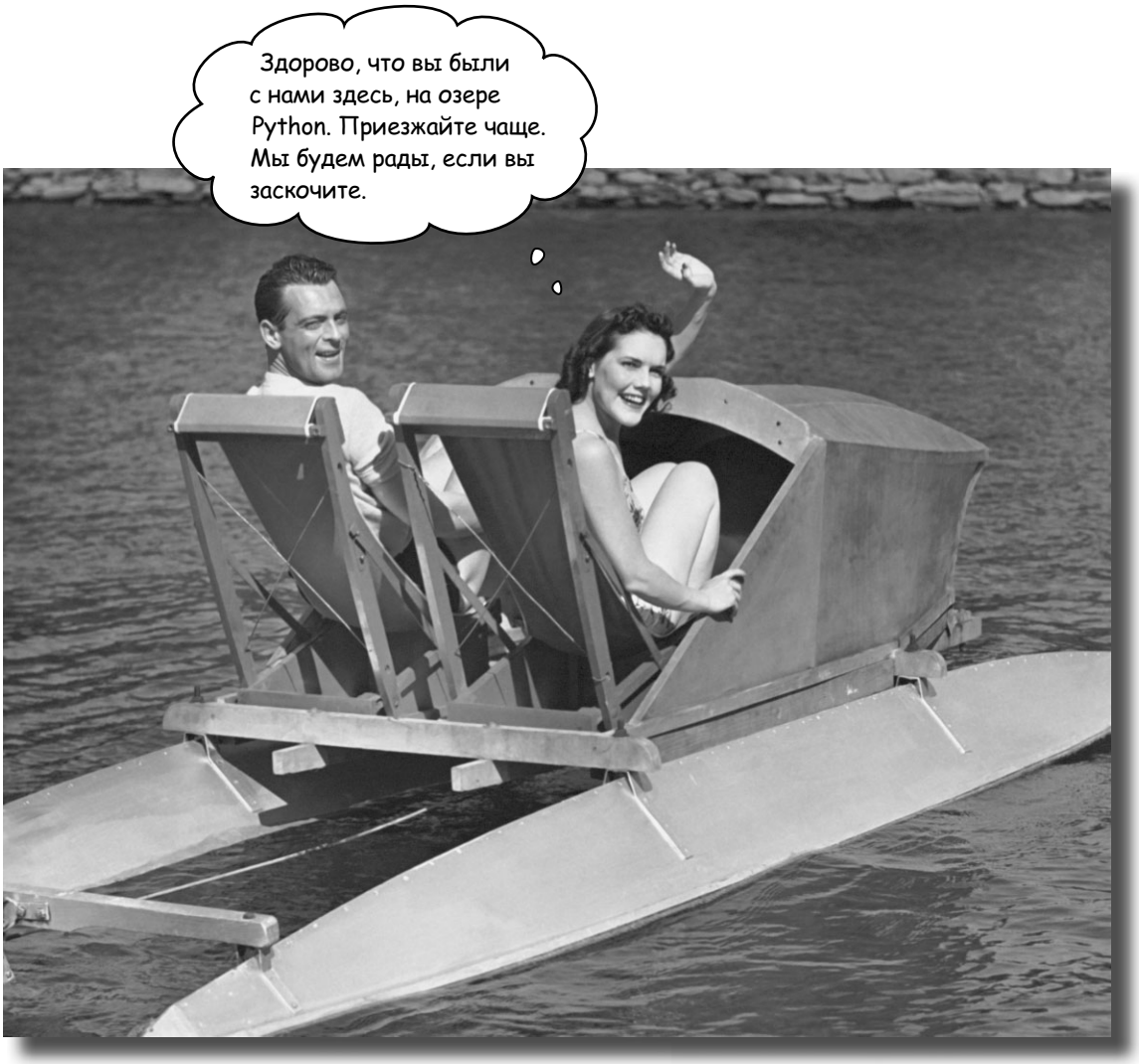
Это «do_convert.py».

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

А это «url_utils.py».

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Пора прощаться...



Счастливого пути!

Нам грустно прощаться, но мы будем счастливы, если вы сможете применить все изученное в этой книге на практике. Вы только начинаете свое путешествие по Python, и вам многое предстоит узнать. Кстати, не торопитесь откладывать эту книгу. Впереди еще пять (да, пять!) приложений, которые нужно прочитать. Обещаем: они не очень длинные, но прочитать их стоит. И, конечно, есть еще алфавитный указатель — давайте не забудем о нем!

Надеемся, вам было нескучно изучать Python, потому что именно такую книгу мы хотели написать. И мы отлично провели время. Наслаждайтесь!

Приложение А: установка

Установка Python

Начнем сначала: установим Python на компьютер.

Какой бы ОС вы ни пользовались — *Windows*, *Mac OS X* или *Linux,* — Python для вас доступен. Порядок установки на каждую из этих платформ зависит от организации процесса в каждой из операционных систем (мы знаем… это звучит пугающе, да?), но разработчики Python хорошо потрудились, создав мастер установки для каждой из популярных систем. В этом коротком приложении вы узнаете, как установить Python на компьютер.

Установка Python 3 в Windows

Если вы (или кто-то другой) не устанавливали интерпретатор Python в своей системе *Windows*, вряд ли он там установлен. Даже если он там есть, давайте установим последнюю версию Python 3 с наибольшим порядковым номером на ваш компьютер с *Windows* на борту.

Если у вас установлена версия Python 3, мастер установки ее обновит. Если у вас установлена версия Python 2, мастер установит версию Python 3 (но она не будет пересекаться с Python 2). Если у вас нет никакой версии Python — что ж, она скоро появится!

Загрузите, потом установите

Откройте в браузере страницу *www.python.org*, затем щелкните на вкладке *Downloads* (Загрузки).

Появятся две большие кнопки, предлагающие выбрать последнюю версию Python 3 или Python 2. *Щелкните на кнопке Python 3.* В ответ на запрос подтвердите желание сохранить файл на компьютере. Немного погодя загрузка файла завершится. Найдите файл в папке *Загрузки* (или где вы его сохранили), затем двойным щелчком мыши на файле запустите установку.

Начнется стандартный процесс установки в *Windows*. По большому счету, вам нужно просто щелкать на кнопке *Next* (Далее) на каждой странице диалога, кроме одной (которая показана ниже), где вам нужно приостановиться и отметить флажок *Add Python 3.5 to Path* (Добавить Python 3.5 в переменную Path); это нужно, чтобы *Windows* смогла легко отыскать интерпретатор, когда потребуется.

Обратите внимание: ко времени выхода этой книги из печати должна появиться новая версия Python 3 (версия 3.6). Поскольку этого не произойдет до 2017 года (который наступит *через* несколько недель после выхода этой книги), мы на всех скриншотах показываем версию 3.5. Не стремитесь, чтобы версия обязательно соответствовала нашей. Загружайте и устанавливайте последний выпуск.

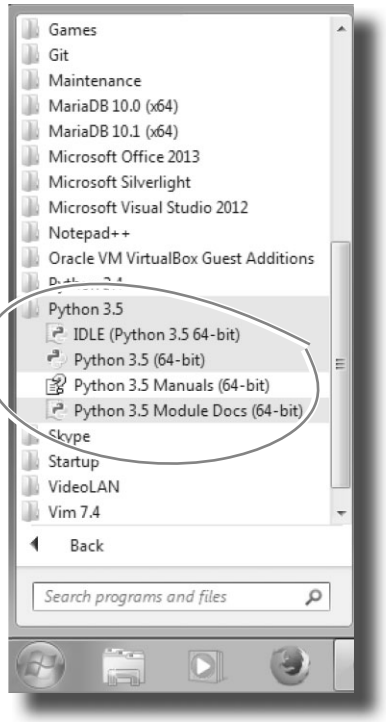
Номер версии, которую вы будете устанавливать, скорее всего, отличается от этого. Не нужно беспокоиться, если вы загрузили более свежую версию — процесс установки аналогичен.

Это действительно важно: установите этот флажок, прежде чем щелкнуть на «Install Now» («Установить сейчас») в этом диалоге.

Проверим правильность установки Python 3 в Windows

После установки интерпретатора Python в *Windows* выполним несколько проверок и убедимся, что все верно.

Для начинающих: у вас должна появиться новая группа в меню *Start* (Пуск), в подменю *All Programs* (Все программы). Мы включили скриншот, показывающий, как выглядит эта группа на одной из машин с *Windows 7* в лаборатории *Head First*. У вас должно получиться нечто подобное. Если нет — попробуйте повторить установку. Пользователи Windows 8 (и более новых версий) также должны обнаружить у себя новую группу, похожую на эту.



Мастер установки Python добавил новую группу в список «All Programs» («Все программы»).

Проверим пункты в группе Python 3.5 снизу вверх.

Пункт *Python 3.5 Modules Docs* (Документация к модулям Python 3.5) открывает доступ к документации, поставляемой с модулями, установленными в вашей системе. Вы многое узнаете о модулях, работая с книгой, поэтому прямо сейчас нет необходимости пользоваться этим пунктом.

Пункт *Python 3.5 Manuals* (Руководства для Python 3.5) содержит исчерпывающую документацию с описанием языка Python в виде стандартного файла справки *Windows*. Это копия документации с описанием Python, доступной в веб.

Пункт *Python 3.5* запускает интерактивную оболочку >>>, которую можно использовать во время разработки для экспериментов с кодом. Нам есть что рассказать о >>>, и уже в главе 1 мы этим займемся. Если вы щелкнули на этом пункте, чтобы проверить, как он работает, а теперь не знаете, что делать, просто наберите `quit()` и вернитесь обратно в *Windows*.

Последний пункт, *IDLE* (*Python 3.5*), запускает интегрированную среду разработки Python, которая называется *IDLE*. Это очень простая IDE, она открывает доступ к командной строке Python >>>, простенькому текстовому редактору, отладчику Python и документации. Мы будем использовать *IDLE* на протяжении всей книги.

Это и есть Python 3 в Windows, в некотором роде...

Чаще всего Python используется в Unix и Unix-подобных системах, и иногда это очень заметно, если вы работаете в *Windows*. Например, некоторое программное обеспечение, которое подразумевает наличие Python, не всегда доступно в *Windows* по умолчанию, поэтому, чтобы получить максимум от Python, программистам в *Windows* часто приходится устанавливать дополнительные компоненты. Давайте установим один из таких компонентов, чтобы продемонстрировать, как добавлять такие недостающие части.

Расширяем набор инструментов Python 3 в Windows

Порой программисты, использующие версию Python для *Windows*, чувствуют себя обделенными: некоторые инструменты, наличие которых предполагается по умолчанию (интерпретатором Python), «отсутствуют» в Windows.

К счастью, для таких случаев написаны сторонние модули, которые можно установить в Python, чтобы восполнить недостающую функциональность. Установка модулей требует совсем немного работы с командной строкой *Windows*.

Для примера добавим в Python популярную утилиту `readline`. Модуль `pyreadline` — Python-версия `readline` — эффективно закрывает этот пробел в конфигурации *Windows* по умолчанию.

Откройте командную строку *Windows* и следуйте за нами. Здесь мы будем использовать инструмент установки модулей (который входит в состав Python 3.5). Этот инструмент называется `pip` (сокр. от «Package Installer for Python» — установщик пакетов для Python).

Для умников

Библиотека `readline` реализует множество функций для интерактивного редактирования текста (обычно в командной строке). Модуль `pyreadline` предоставляет интерфейс к `readline`.

В командной строке Windows введите ***`pip install pyreadline`***.

Вот что вам нужно набрать в командной строке.

```
File  Edit  Window  Help  InstallingPyReadLine
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Head First>
C:\Users\Head First> pip install pyreadline
Downloading/unpacking pyreadline
      ...
      ...
      ...
Successfully installed pyreadline
Cleaning up...
C:\Users\Head First>
```

Здесь вы увидите довольно много сообщений.

Если видите это сообщение — все ОК.

Убедитесь, что у вас есть подключение к Интернету, прежде чем выполнить эту команду.

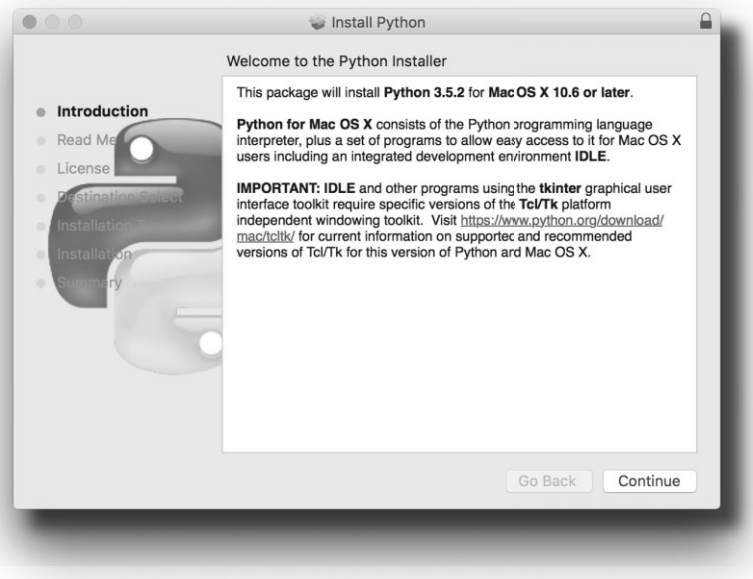
Теперь пакет `pyreadline` установлен и готов к работе.

Можете перейти к главе 1 и начать выполнять код на Python из примеров.

Установка Python 3 в Mac OS X (macOS)

В *Mac OS X* по умолчанию предустановлен Python 2. Но для нас он бесполезен, потому что вместо него мы будем использовать Python 3. К счастью, сайт проекта Python (*http://www.python.org*) в состоянии распознать, что вы используете Mac. Выберите вкладку *Downloads* (Загрузки), затем щелкните на кнопке 3.5.x, чтобы загрузить дистрибутив Python для Mac. Выберите последнюю версию Python 3, загрузите дистрибутив и установите его обычным для Mac способом.

Стандартный мастер установки Python 3.5.2 и выше для Mac OS X. Если есть более свежая версия, чем та, что мы показываем, это еще лучше — устанавливайте!



Просто щелкайте, пока установка не будет завершена.

Использование диспетчера пакетов

В Mac также есть возможность использовать один из популярных *диспетчеров пакетов* с открытым исходным кодом, например *Homebrew* или *MacPorts*. Если вы никогда не пользовались ни одним из этих диспетчеров пакетов, можете пропустить этот короткий раздел и перейти к следующей странице. Но если вы уже использовали какой-либо из них, здесь вы найдете команды, которые нужны для установки Python 3 в Mac из окна терминала.

- Для *Homebrew*, введите **`brew install python3`**.
- Для *MacPorts*, введите **`port install python3`**.

Вот и все: вы в шоколаде. Python3 готов к использованию в *Mac OS X* — давайте посмотрим, что же было установлено.

Проверка и настройка Python 3 в Mac OS X

Чтобы убедиться, что установка в *Mac OS X* прошла успешно, щелкните на значке *Applications* (Приложения) в панели инструментов, затем загляните в папку *Python 3*.

Щелкните на папке *Python 3*, и вы увидите группу значков (см. ниже).

Папка Python 3 в Mac OS X

Первый значок, *IDLE*, один из наиболее полезных, с его помощью вы будете взаимодействовать с Python 3 большую часть времени при изучении языка. Этот значок открывает интегрированную среду разработки Python, которая называется *IDLE*. Это очень простая IDE, открывающая доступ к интерактивной оболочке Python >>>, простенькому текстовому редактору, отладчику Python и документации. Мы будем достаточно часто использовать *IDLE* на протяжении всей книги.

Значок *Python Documentation.html* открывает в браузере локальную копию документации в формате HTML с описанием языка Python (не требует подключения к Сети).

Значок *Python Launcher* автоматически запускается системой *Mac OS X*, если выполнить двойной щелчок на выполняемом файле, содержащем код на Python. Кому-то этот значок может пригодиться. Мы редко пользуемся им в лаборатории *Head First Labs*, однако все равно приятно знать, что он есть и мы сможем им воспользоваться, если вдруг понадобится.

Последний значок, *Update Shell Profile.command*, обновляет конфигурационные файлы в *Mac OS X*, чтобы местоположение интерпретатора Python и всех его компонентов было корректно прописано в переменной `PATH` операционной системы. Вы можете щелкнуть на этом значке прямо сейчас и запустить команду, а затем забыть о нем — одного раза достаточно.

Папка Python 3 внутри папки Applications (Приложения) в Mac OS X.

Щелкните на значке Applications (Приложения) в панели инструментов, а затем выберите папку Python 3.

Вы готовы к запуску в Mac OS X

Теперь у вас есть полный набор инструментов для работы в *Mac OS X*.

Можете переходить к главе 1 и начинать изучение.

Установка Python 3 в Linux

Если вы пользуетесь самой свежей версией вашего любимого дистрибутива *Linux*, спешим вас *обрадовать*: скорее всего, в вашей системе уже установлены Python 2 *и* Python 3.

Вот самый простой способ узнать текущую версию установленного интерпретатора Python; откройте командную строку и наберите:

```
$ python3 -V
3.5.2
```

Будьте внимательны: «V» здесь в ВЕРХНЕМ РЕГИСТРЕ

Как круто, да? В нашем Linux уже установлена последняя версия Python 3.

Если после ввода этой команды *Linux* сообщил, что не может найти программу `python3`, вам нужно ее установить. Как это сделать — зависит от дистрибутива Linux, которым вы пользуетесь.

Если ваш *Linux* основан на популярных дистрибутивах *Debian* или *Ubuntu* (как тот, что мы используем в *Head First Labs*), вы можете использовать для установки Python 3 утилиту `apt-get`.

```
$ sudo apt-get install python3 idle3
```

Если вы пользуетесь дистрибутивом с диспетчером пакетов *yum* или *rpm*, используйте эквивалентную команду. Или запустите графический интерфейс и используйте графический диспетчер пакетов, чтобы отыскать и установить `python3` *и* `idle3`. Многие системы *Linux* используют *Synaptic Package Manager*, а также другие утилиты установки с графическим интерфейсом.

Выберите пакеты «python3» и «idle3» для установки в Linux.

После установки Python 3 проверьте правильность настройки командой, приведенной в начале страницы.

Вне зависимости от того, какой дистрибутив вы используете, команда `python3` откроет доступ к интерпретатору, а `idle3` запустит графическую интегрированную среду IDLE. Это очень простая IDE, которая открывает доступ к интерактивной оболочке Python >>>, простенькому текстовому редактору, отладчику Python и документации.

Мы будем использовать консоль >>> и IDLE начиная с главы 1, к которой вы можете перейти прямо сейчас.

Приложение B: PythonAnywhere

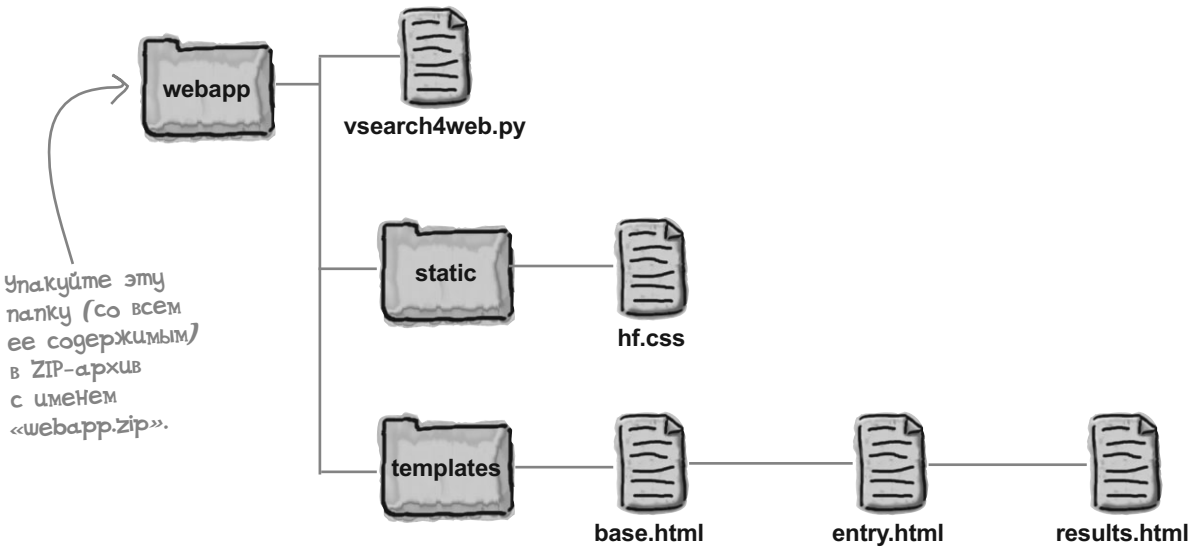
Развертывание веб-приложения

В конце главы 5 мы говорили, что развертывание веб-приложения в облаке займет всего лишь 10 минут.

Пора выполнить обещание. В этом приложении мы собираемся провести вас через весь процесс развертывания веб-приложения в *PythonAnywhere* за 10 минут с самого начала до полностью развернутого приложения. *PythonAnywhere* — это служба, популярная в сообществе программирования на Python, и нетрудно понять почему: она работает в полном соответствии с ожиданиями, имеет отличную поддержку Python (и Flask) и — самое важное — позволяет начать с бесплатного размещения веб-приложения. Давайте оценим *PythonAnywhere*.

Шаг 0: Небольшая подготовка

Сейчас у нас есть код веб-приложения, находящийся в папке `webapp`, которая содержит файл `vsearch4web.py`, а также папки `static` и `templates` (как показано ниже). Чтобы подготовить все это к развертыванию, создайте ZIP-архив с папкой `webapp` и дайте ему имя `webapp.zip`.



Кроме `webapp.zip` также нужно выгрузить и установить модуль `vsearch` из главы 4. Для этого нужно просто найти файл дистрибутива, который вы тогда создали. На нашем компьютере файл архива называется `vsearch-1.0.tar.gz` и хранится в папке `mymodules/vsearch/dist` (в *Windows* этот файл, вероятно, будет называться `vsearch- 1.0.zip`).

Если вы еще помните главу 4, модуль «setuptools» в Python создает в Windows файлы ZIP и tar.gz на всех других платформах.

Прямо сейчас вам ничего не нужно делать с файлом архива. Просто отметьте, где находятся оба архива на вашем компьютере, чтобы их можно было легко найти, когда будете выгружать их в *PythonAnywhere*. Возьмите карандаш и запишите здесь местонахождение каждого файла.

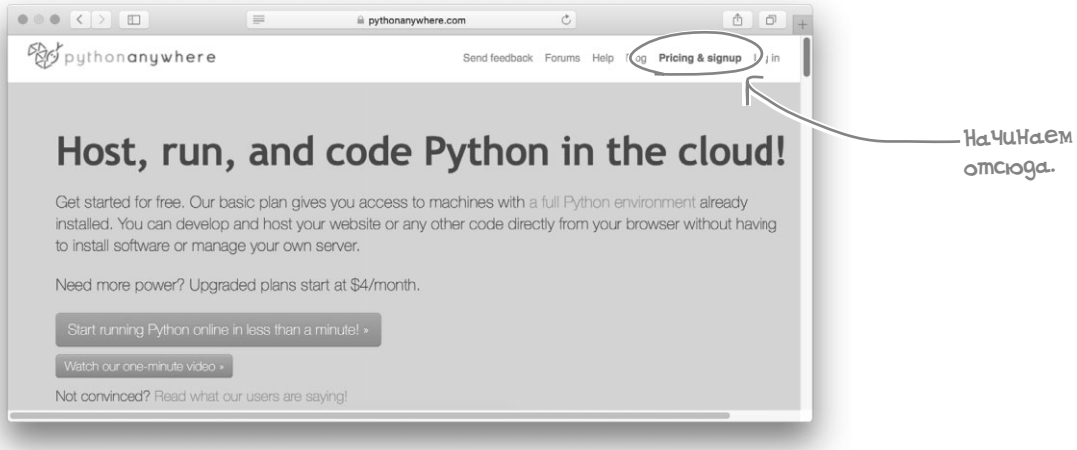
webapp.zip _____

vsearch-1.0.tar.gz _____

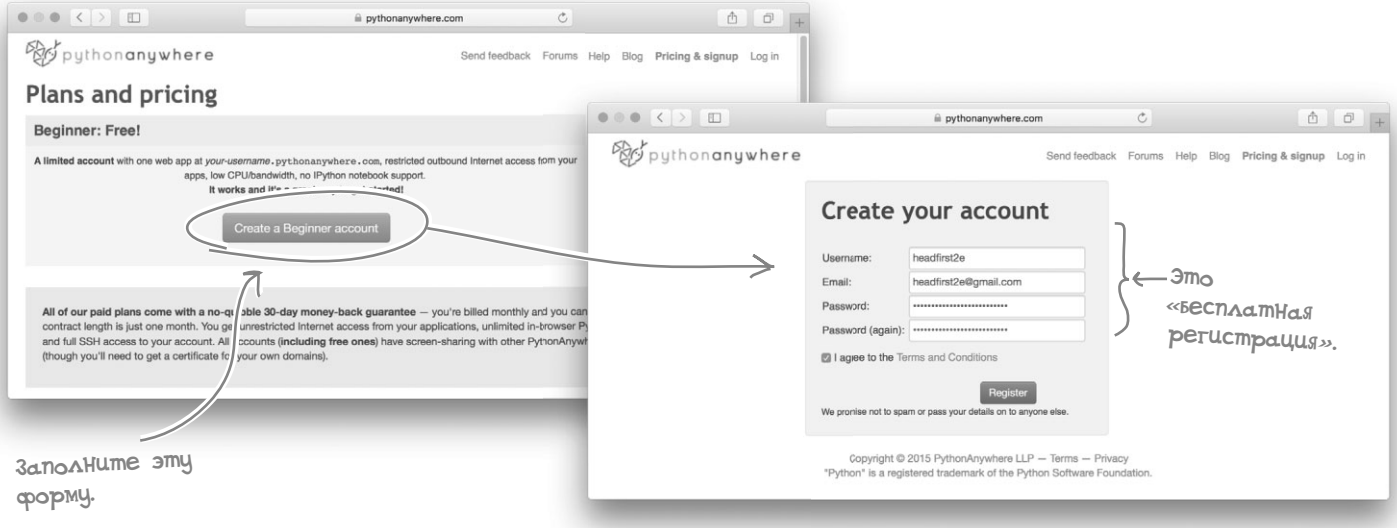
Если вы в Windows, тут должно быть имя «vsearch.zip».

Шаг 1: Регистрируемся в PythonAnywhere

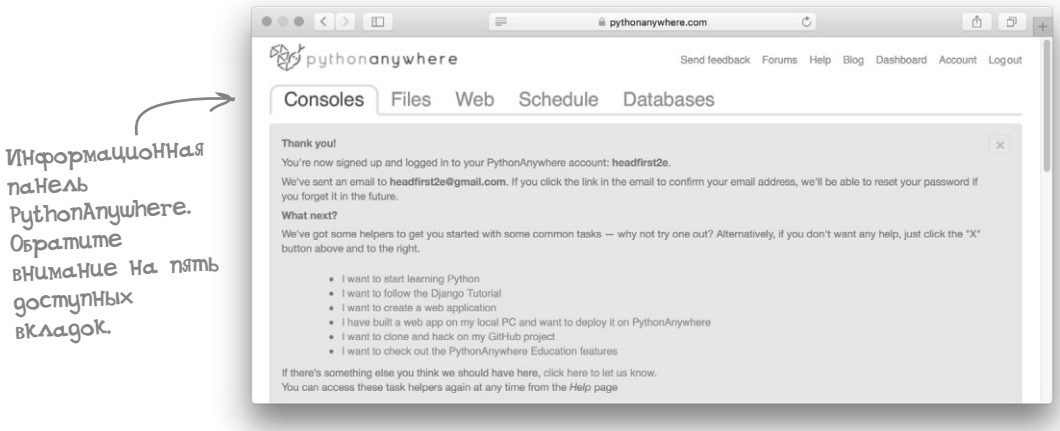
Этот шаг будет, вероятно, самым простым. Откройте страницу *pythonanywhere.com*, щелкните по ссылке **Pricing & signup** (Цены и регистрация).



Щелкните на большой синей кнопке, чтобы создать *учетную запись начинающего*, а затем заполните регистрационную форму.

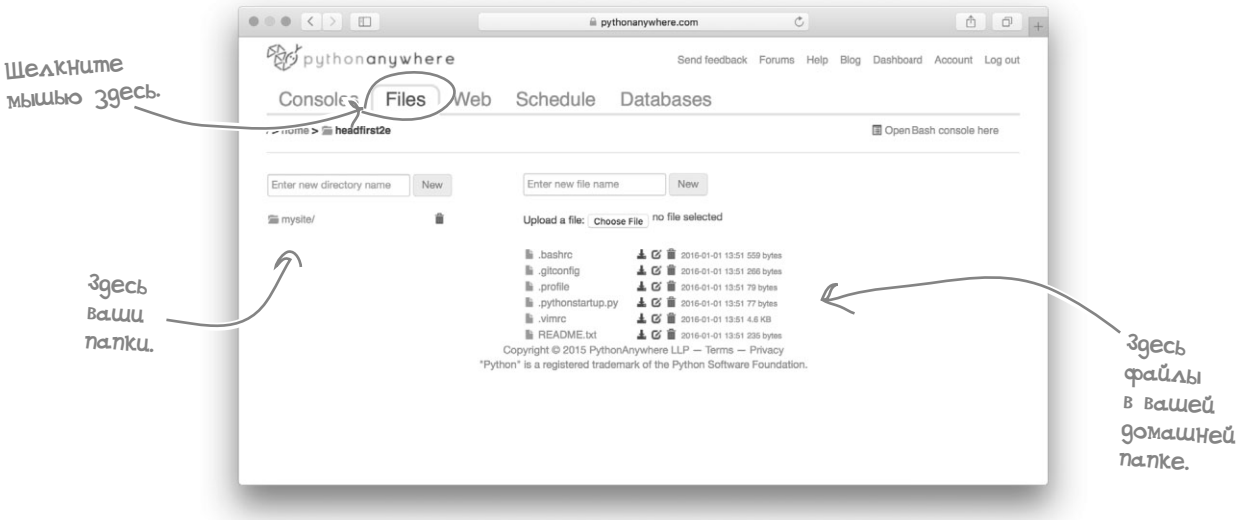


Если все пройдет успешно, появится информационная панель PythonAnywhere. Обратите внимание: в этот момент вы уже зарегистрированы и вошли в систему.

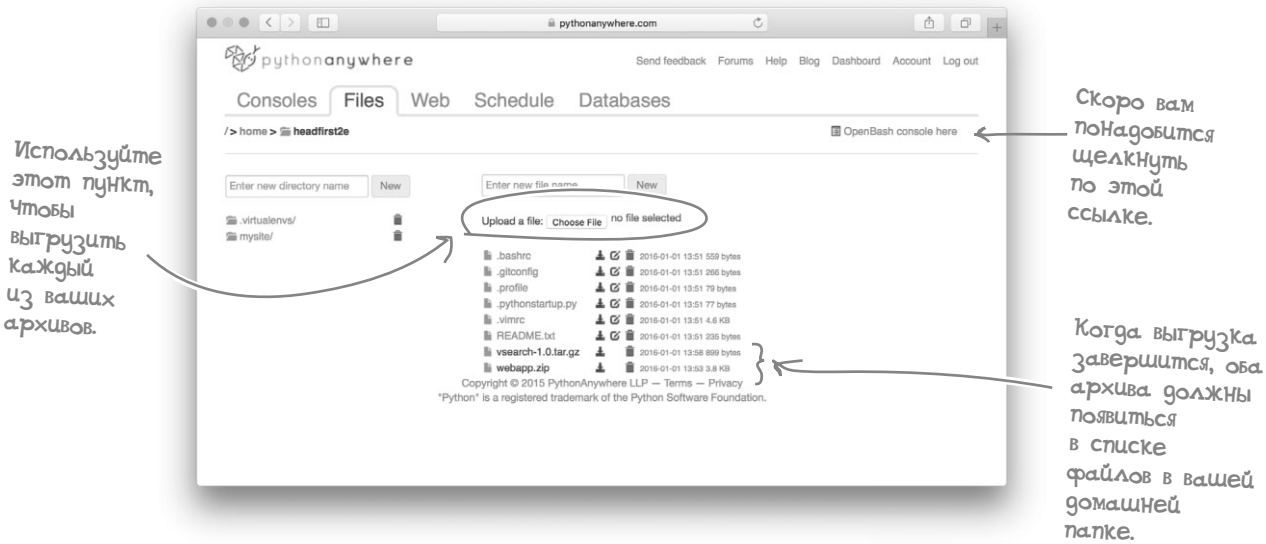


Шаг 2: Выгружаем файлы в облако

Щелкните мышкой на вкладке **Files** (Файлы), чтобы посмотреть доступные вам папки и файлы.



Используйте пункт *Upload a file* (Выгрузить файл), чтобы выбрать и выгрузить два архива из **шага 0**.



Теперь вы готовы распаковать и установить выгруженные архивы, и делать это вы будете в ходе выполнения **шага 3**. Чтобы перейти к нему, щелкните по ссылке *Open a bash console here* (Открыть командную строку здесь) вверху справа на странице, изображенной выше. Она откроет окно терминала в окне браузера (в *PythonAnywhere*).

Шаг 3: Извлекаем и устанавливаем код

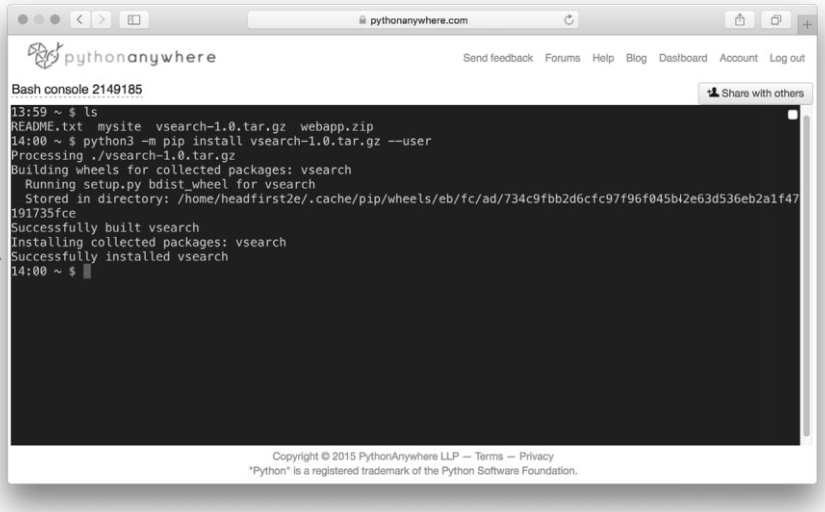
После щелчка по ссылке *Open a bash console here* (Открыть командную строку здесь) *PythonAnywhere* заменит информационную панель *Files* (Файлы) версией консоли (командной строки) Linux для браузеров. Выполните несколько команд в этой консоли, чтобы извлечь и установить модуль `vsearch`, а также код веб-приложения. Сначала установите `vsearch` в Python как «частный модуль» (то есть только для себя), используя эту команду (используйте имя `vsearch-1.0.zip`, если вы в *Windows*).

```
python3 -m pip install vsearch-1.0.tar.gz --user
```

«–user» гарантирует, что модуль «vsearch» будет установлен только для вас. PythonAnywhere не позволит установить модуль для всеобщего использования (только для личного).

Запуск команды.

Успех!



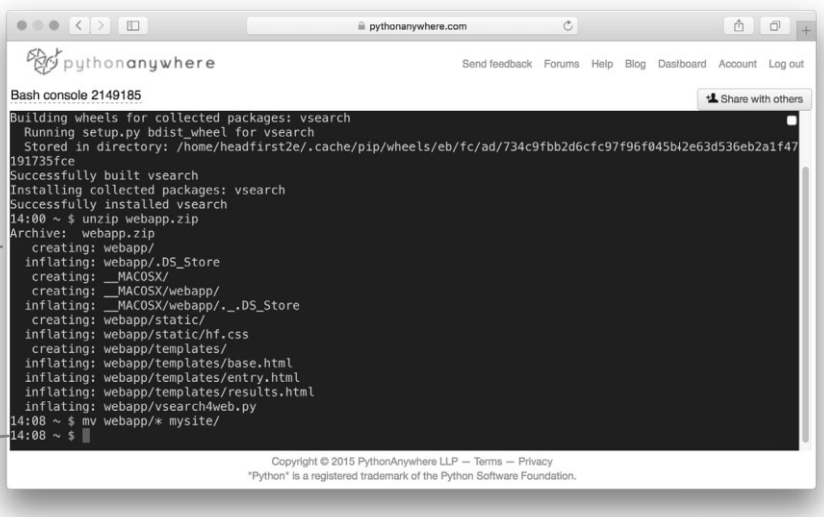
После успешной установки модуля `vsearch` наступает черед кода веб-приложения, который нужно установить в папку `mysite` (она уже присутствует в вашей домашней папке *PythonAnywhere*). Чтобы сделать это, выполните две команды.

Распакуйте код веб-приложения...

```
unzip webapp.zip
mv webapp/* mysite
```

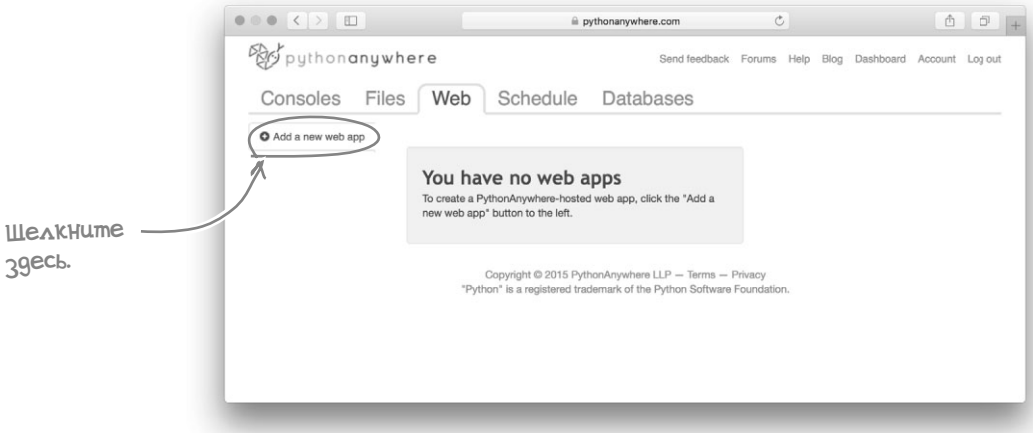
...затем переместите код в папку «mysite».

Вы должны увидеть точно такие же сообщения.

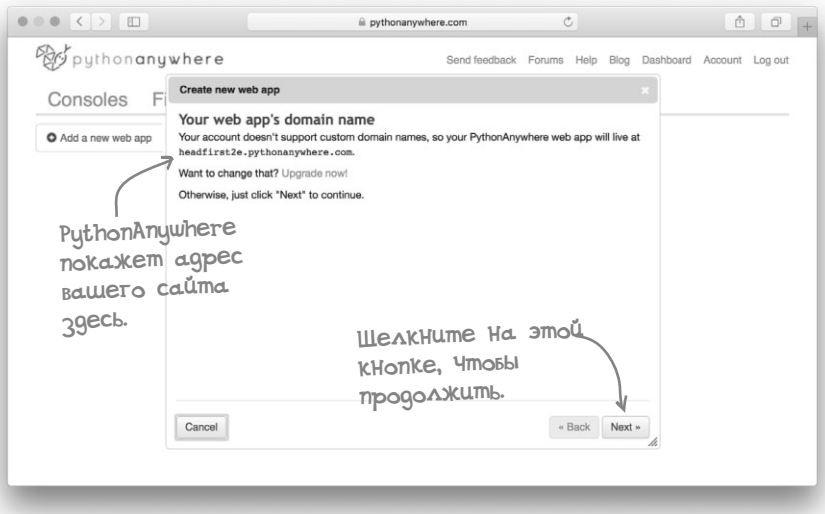


Шаг 4: Создаем начальное веб-приложение, 1 из 2

Выполнив **шаг 3**, вернитесь в информационную панель *PythonAnywhere* и выберите вкладку **Web**, где *PythonAnywhere* пригласит вас создать новое начальное веб-приложение. Сделайте это, а затем замените код веб-приложения своим собственным. Заметьте, что каждая *учетная запись начинающего* позволяет бесплатно развернуть только одно веб-приложение; если вы хотите больше, вам придется обновить ее до платной учетной записи. К счастью, на данный момент нам нужно только одно приложение, так что давайте продолжим, щелкнув на кнопке *Add a new web app* (Добавить новое веб-приложение):



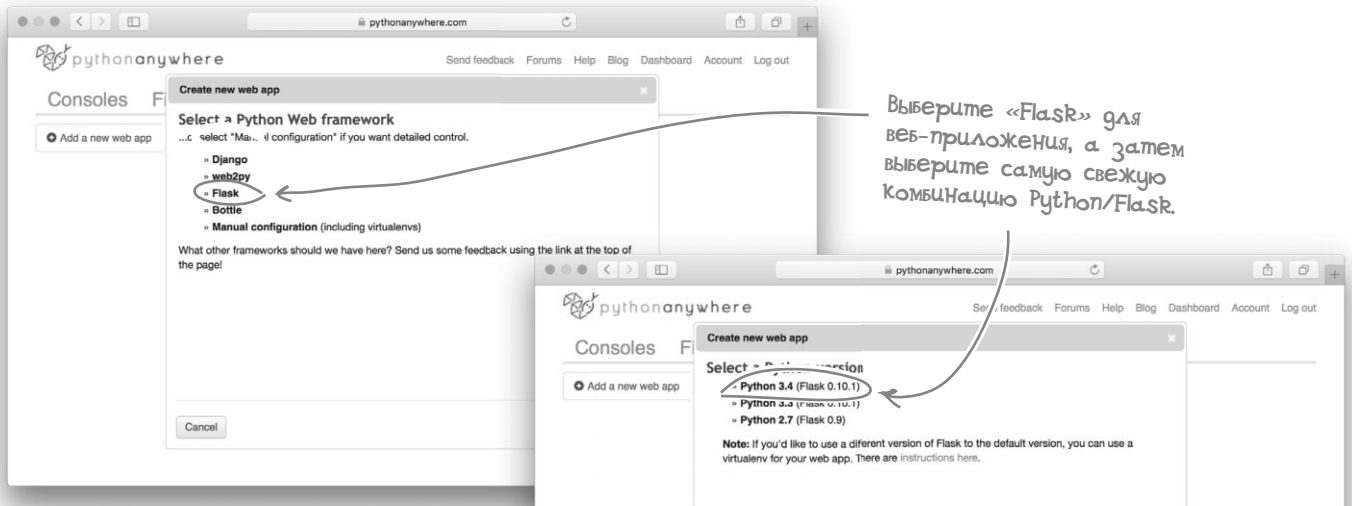
Так как используется бесплатная учетная запись, веб-приложение будет доступно по адресу, показанному на следующем экране. Щелкните на кнопке *Next* (Далее), чтобы принять адрес сайта, предлагаемый службой *PythonAnywhere*.



Для продолжения щелкните по кнопке *Next* (Далее).

Шаг 4: Создаем начальное веб-приложение, 2 из 2

PythonAnywhere поддерживает несколько веб-фреймворков на Python, поэтому на следующем экране вам предложат выбрать нужный из множества поддерживаемых. Выберите Flask, а затем выберите версию Flask и Python, которые вы желаете развернуть. В момент написания этих строк самыми свежими версиями, которые поддерживала служба *PythonAnywhere*, были Python 3.4 и Flask 0.10.1, так что продолжите с этой комбинацией, если не будет предложено ничего свежée (или выберите наиболее свежие версии).

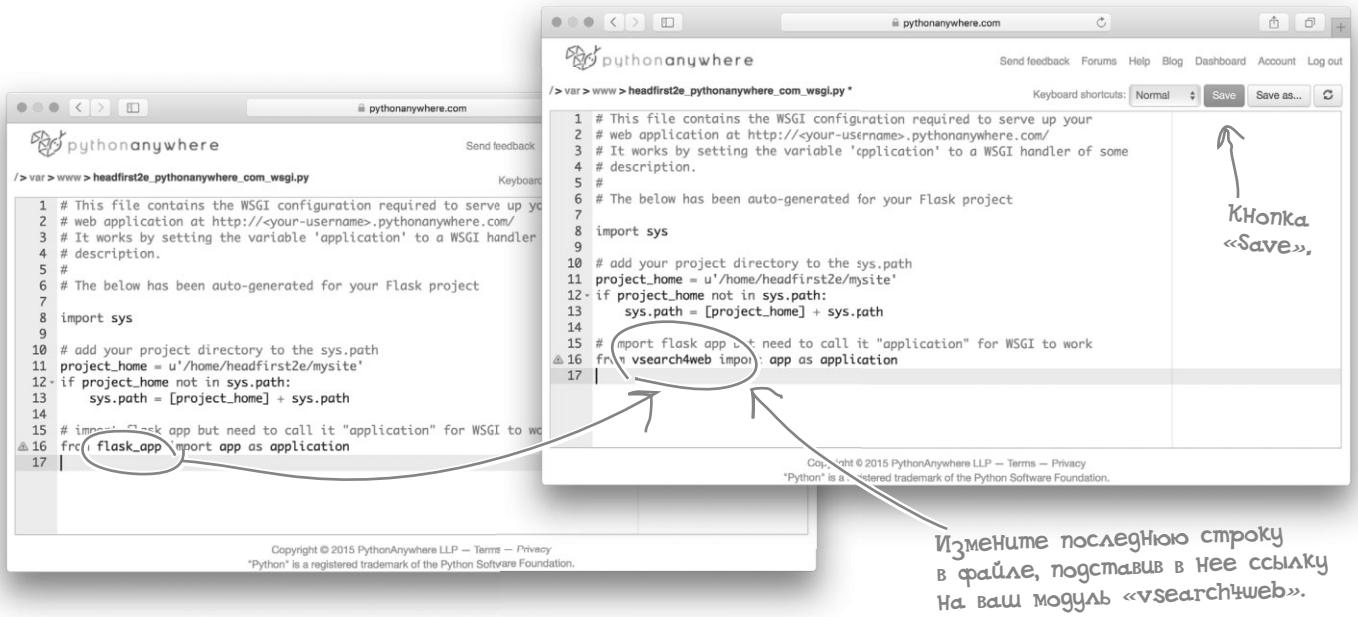


Следующий экран предложит создать начальное веб-приложение с поддержкой Flask. Продолжайте и сделайте это сейчас, подтвердив согласие с предложенными значениями щелчком на кнопке *Next* (Далее).

Шаг 5: Настраиваем веб-приложение

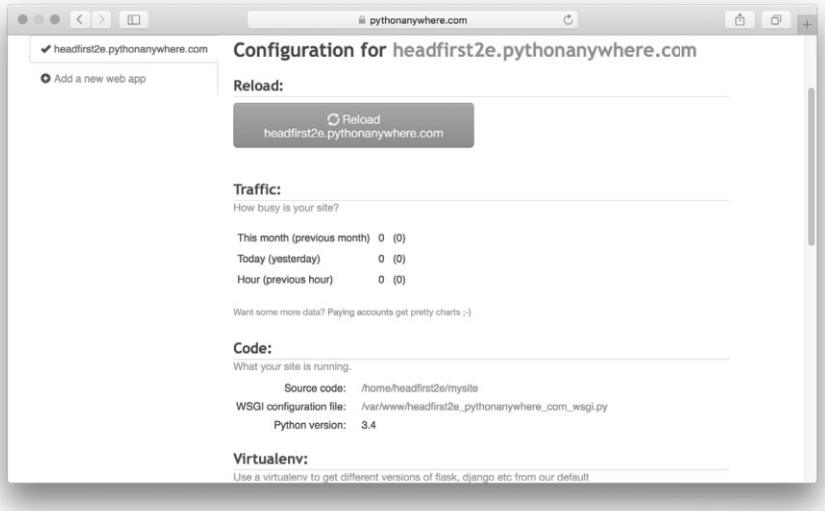
После завершения **шага 4** откроется вкладка **Web** в информационной панели. Не поддавайтесь искушению щелкнуть мышью на этой большой зеленой кнопке — вы пока не передали службе *PythonAnywhere* свой код, так что сейчас отложите запуск чего бы то ни было. Вместо этого щелкните на длинной ссылке справа: *WSGI configuration file* (Конфигурационный файл WSGI).

Щелчок на этой длинной ссылке загрузит конфигурационный файл вновь созданного веб-приложения в текстовый веб-редактор *PythonAnywhere*. В конце главы 5 мы сказали, что *PythonAnywhere* автоматически импортирует код веб-приложения, прежде чем вызвать `app.run()`. Это файл, который описывает такое поведение. Однако в нем нужно указать ссылку на *ваш* код, а не на код начального приложения, поэтому отредактируйте последнюю строку в файле (как показано ниже), а затем щелкните на кнопке *Save* (Сохранить).

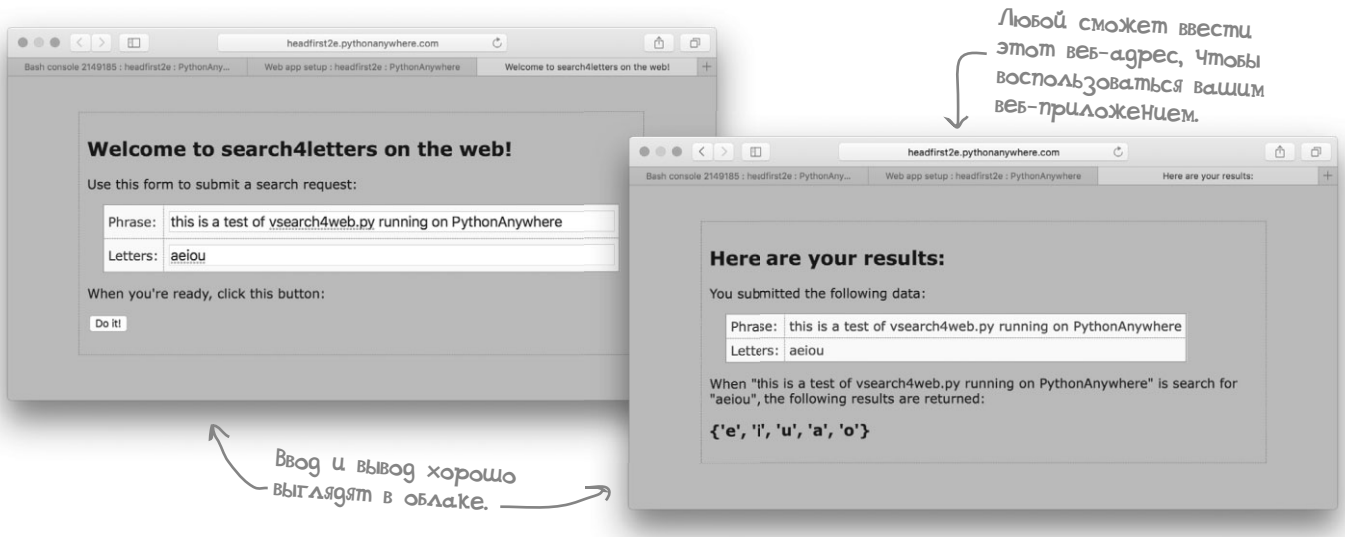


Шаг 6: Запускаем облачное веб-приложение!

Не забудьте сохранить измененный конфигурационный файл, а затем вернитесь на вкладку **Web** в информационной панели. Вот теперь пора щелкнуть на этой большой, заманчивой, зеленой кнопке. Сделайте это!



Через короткий промежуток времени ваше веб-приложение появится в браузере и будет работать в точности, как это было, когда мы запускали его локально, но теперь его сможет использовать любой, у кого есть подключение к Интернету и веб-браузер.



Вот и все. Веб-приложение, которое было разработано в главе 5, развернуто в облаке *PythonAnywhere* (менее чем за 10 минут). В *PythonAnywhere* есть намного больше, чем было показано в этом коротком приложении, так что не стесняйтесь исследовать и экспериментировать. Наигравшись, не забудьте вернуться в информационную панель *PythonAnywhere* и выйти из системы. Обратите внимание, что после выхода из системы веб-приложение останется запущенным в облаке, пока вы не остановите его явно. Круто, правда?

Приложение C: топ-10 тем, которые мы не рассмотрели

Всегда есть чему поучиться

Мы и не стремились рассмотреть все.

Наша цель — познакомить вас с Python, чтобы вы как можно скорее могли приступить к работе. Мы о многом умолчали. В этом приложении мы обсудим топ-10 тем, которые — будь у нас еще 600 страниц или около того — мы не обошли бы вниманием. Не все 10 пунктов будут вам интересны, но вы можете быстро пролистать их — вдруг мы затронули интересную для вас тему или дали ответ на насущный вопрос. Все технологии программирования в приложении готовы к использованию в Python и его интерпретаторе.

1. Что насчет Python 2?

На момент публикации этой книги (конец 2016 года) существовали две основных и самых популярных версии Python. Вы уже знаете немного о **Python 3**, поскольку эта версия использовалась в данной книге.

Все новшества и улучшения вносятся в Python 3, минорные версии которого появляются каждые 12–18 месяцев. Версия 3.6 была выпущена в конце 2016 года, и можно ожидать, что версия 3.7 будет выпущена в конце 2017 или в начале 2018 года.

Python 2 некоторое время назад «застрял» на версии 2.7. Это означает, что *ведущие разработчики Python* (люди, которые руководят разработкой Python) решили, что будущее за Python 3 и Python 2 должен тихо уйти на покой. Это решение было принято под давлением веских технических причин, но на самом деле никто не ожидал, что все это будет тянуться так долго. В конце концов, Python 3 — будущее языка — впервые появился в конце 2008 года.

Можно было бы написать целую книгу о том, что произошло с конца 2008-го до настоящего момента. Достаточно сказать, что Python 2 упрямо отказывается уходить со сцены. Было (и остается) огромное количество действующего кода на Python 2 и разработчиков, которые не торопятся переходить на новую версию. Есть очень простая причина, почему так происходит: Python 3, представляя горстку улучшений, ломает обратную совместимость. Перефразируем: есть множество кода на Python 2, который не будет запускаться *в неизмененном виде* в Python 3 (даже притом что, на первый взгляд, трудно отличить код на Python 2 от кода на Python 3). Также множество программистов просто уверовали, что Python 2 был «достаточно хорош», и поэтому не проводили модернизацию.

В последнее время (за последний год) произошло море изменений. Скорость перехода с версии 2 на 3, кажется, стала увеличиваться. Появились версии некоторых очень популярных сторонних модулей, совместимые с Python 3, и это возымело положительный эффект для признания Python 3. В дополнение к этому *ведущие разработчики Python* продолжают добавлять дополнительные полезные штуки в Python 3, постоянно увеличивая привлекательность этого языка программирования. Практика «обратного переноса» новых крутых возможностей из версии 3 в версию 2 была прекращена на версии 2.7, и хотя исправления ошибок и проблем безопасности все еще вносятся, *ведущие разработчики Python* объявили, что эта активность в 2020 году будет остановлена. Время Python 2 истекает.

Вот общий совет, который мы можем дать всем, кто выбирает между версиями 2 и 3:

Если вы начинаете новый проект, используйте Python 3.

Сопротивляйтесь попыткам побудить вас писать код на устаревшей версии Python 2, особенно если вы начинаете с чистого листа. Если вам придется сопровождать существующий код на Python 2, полученных знаний о версии 3 будет вполне достаточно: вы, безусловно, сможете читать код и понимать его (поскольку это все еще Python, несмотря на другой старший номер версии). Если есть технические причины, чтобы код продолжал работать в Python 2, то так тому и быть. Однако если код можно без особой суеты перенести на Python 3, тогда, как нам кажется, эту боль стоит перетерпеть, потому что Python 3 *лучше*, и будущее *за ним*.

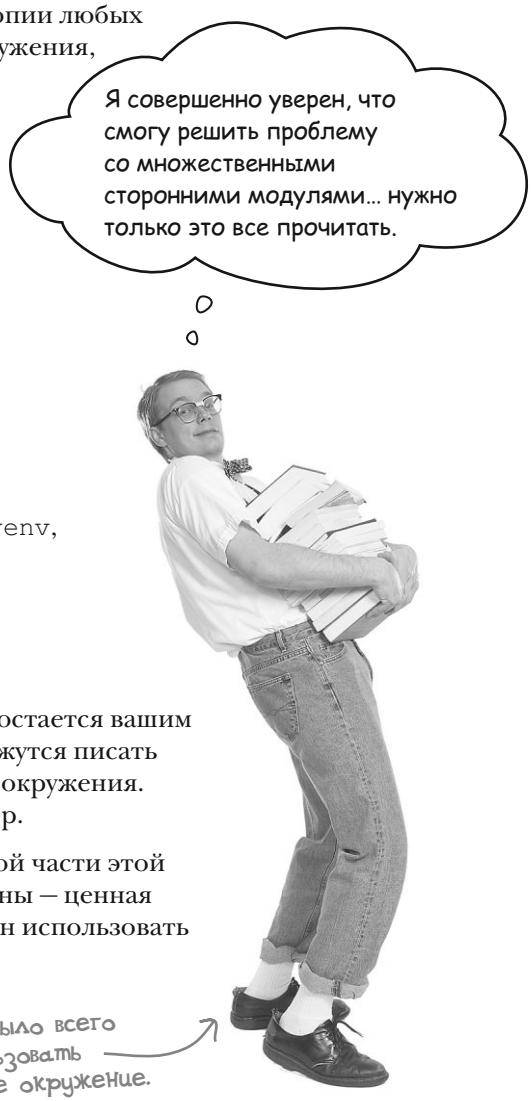
2. Виртуальное программное окружение

Давайте представим, что у вас есть два клиента, один с кодом на Python, который полагается на одну версию стороннего модуля, а код другого полагается на *другую* версию того же модуля. И вы — та самая бедная душа, которая сопровождает код обоих проектов.

Сделать это на одном компьютере может быть проблематично, потому что интерпретатор Python не поддерживает установку сразу нескольких версий сторонних модулей.

Тем не менее помощь всегда под рукой, благодаря такому понятию Python, как виртуальное окружение.

Виртуальное окружение позволяет создать новое, чистое окружение Python, внутри которого вы сможете запускать свой код. Вы можете устанавливать сторонние модули в одно виртуальное окружение, никоим образом не воздействуя на другие, и иметь столько виртуальных окружений, сколько позволит компьютер, переключаясь между ними посредством *активации* необходимого. Так как каждое виртуальное окружение может содержать свои копии любых сторонних модулей, можно создать два разных виртуальных окружения, по одному для каждого из проектов, упомянутых выше.



Однако прежде вы должны будете выбрать между технологией виртуальных окружений, которая называется `venv` и входит в состав *стандартной библиотеки* Python 3, и сторонним модулем `virtualenv` из PyPI (который делает все то же, что и `venv`, но имеет больше бантиков и рюшечек). Будет лучше, если вы сделаете информационно обоснованный выбор.

Узнать больше о `venv` можно в документации по адресу:

```
https://docs.python.org/3/library/venv.html
```

Чтобы узнать, что предлагает `virtualenv` сверх того, что дает `venv`, начните отсюда:

```
https://pypi.org/project/virtualenv/
```

Использовать ли виртуальное окружение для ваших проектов — остается вашим личным выбором. Некоторые программисты клянутся, что откажутся писать какой-либо код на Python, пока он не будет внутри виртуального окружения. Возможно, это чересчур радикальная позиция, но таков их выбор.

Мы решили не рассматривать виртуальные окружения в основной части этой книги. Мы считаем, что виртуальные окружения — если они нужны — ценная находка, но не верим, что каждый программист на Python должен использовать их для всего, что он делает.

Наш совет: обходите стороной тех, кто говорит, что вы не настоящий программист на Python, *если не* используете `virtualenv`.

3. Больше объектной ориентированности

Если вы прочитали книгу полностью, то сейчас (надеемся на это) оцените, что означает фраза: «в Python все является объектом».

Поддержка объектов в Python продумана до мелочей. То есть обычно все работает именно так, как ожидается. Но тот факт, что все является объектом, **не** означает, что все принадлежит классу, особенно когда дело доходит до вашего кода.

Мы не касались темы создания собственных классов, пока не возникла необходимость реализовать свой диспетчер контекста. Но даже тогда мы изучили только самое необходимое, и ничего сверх того. Если вы переходите на Python из языка программирования, который требует, чтобы весь код находился в классах (типичным примером может служить *Java*), тот способ, которым мы воспользовались, может привести в замешательство. Но пусть вас это не беспокоит, так как Python гораздо менее строг, чем (например) *Java*, в отношении организации программ.

Если вам хочется создать целую кучу функций, чтобы выполнить необходимую работу, то просто сделайте это. Если вы мыслите в функциональной манере, Python пригодится вам своим синтаксисом генераторов и даже приподнимет шляпу в знак уважения функциональному программированию. И если вы вынуждены поместить свой код в классы, Python даст вам полноценный синтаксис объектно-ориентированного программирования.

Итак, если вы тратите много времени на создание классов, вам определенно пригодятся следующие элементы.

- `@staticmethod`: декоратор, позволяющий создать статическую функцию внутри класса (без первого параметра `self`).
- `@classmethod`: декоратор, позволяющий создать метод класса, принимающий в первом параметре (как правило, с именем `cls`) сам класс вместо ссылки `self` на объект.
- `@property`: декоратор, позволяющий определять и использовать метод, как если бы он был атрибутом.
- `__slots__`: директива класса может (при использовании) значительно повысить эффективность использования памяти объектами, созданными из вашего класса (за счет некоторой потери гибкости).

Узнать больше о любом из этих элементов можно в документации Python (*https://docs.python.org/3/*).

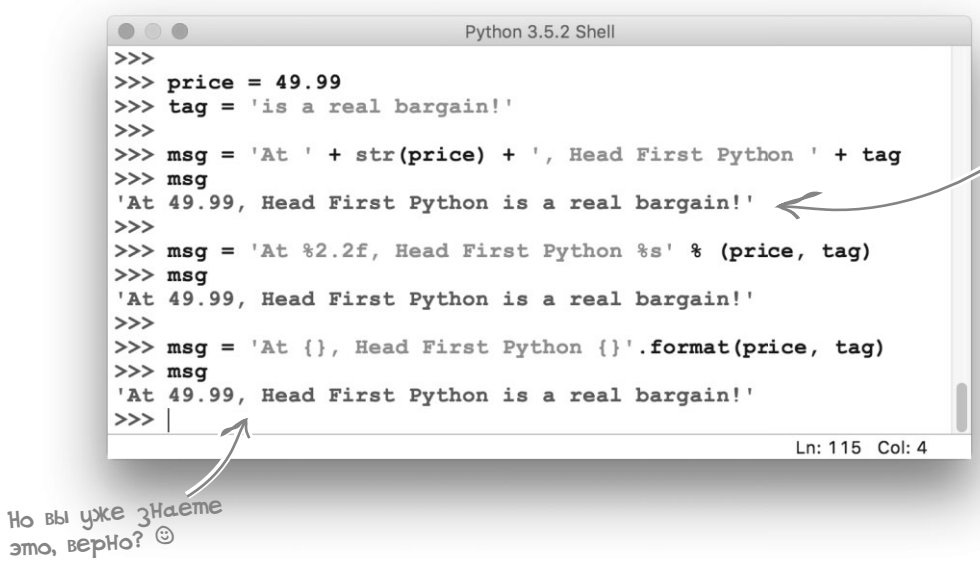
4. Форматирование строк и тому подобное

В разных главах книги мы использовали пример приложения, которое выводило результаты своей работы в веб-браузер. Это позволило нам переложить форматирование вывода на HTML (в частности, если мы использовали модуль *Jinja2*, включенный в фреймворк *Flask*). При этом мы обошли стороной одну из областей, где Python блистает во всей красе: форматирование текстовых строк.

Скажем, вам нужно создать строку, содержащую значения, которые неизвестны до момента выполнения кода. Вы хотите создать сообщение (`msg`), включающее значения, для использования где-то еще (возможно, чтобы вывести его на экран, добавить внутрь страницы HTML, которая создается с использованием *Jinja2*, или твитнуть сообщение трем миллионам своих подписчиков). Значения генерируются вашим кодом во время выполнения и хранятся в двух переменных: `price` (цена предмета, о котором идет речь) и `tag` (броский маркетинговый слоган). У вас на выбор есть несколько вариантов.

- Сформировать сообщение, используя операцию конкатенации строк (оператор +).
- Использовать старый стиль форматирования строк (с применением %-синтаксиса).
- Использовать метод строк `format` для формирования сообщения.

Следующий короткий сеанс в интерактивной оболочке >>> показывает все три приема (сейчас, после чтения книги, вы наверняка согласны с нашим сообщением).



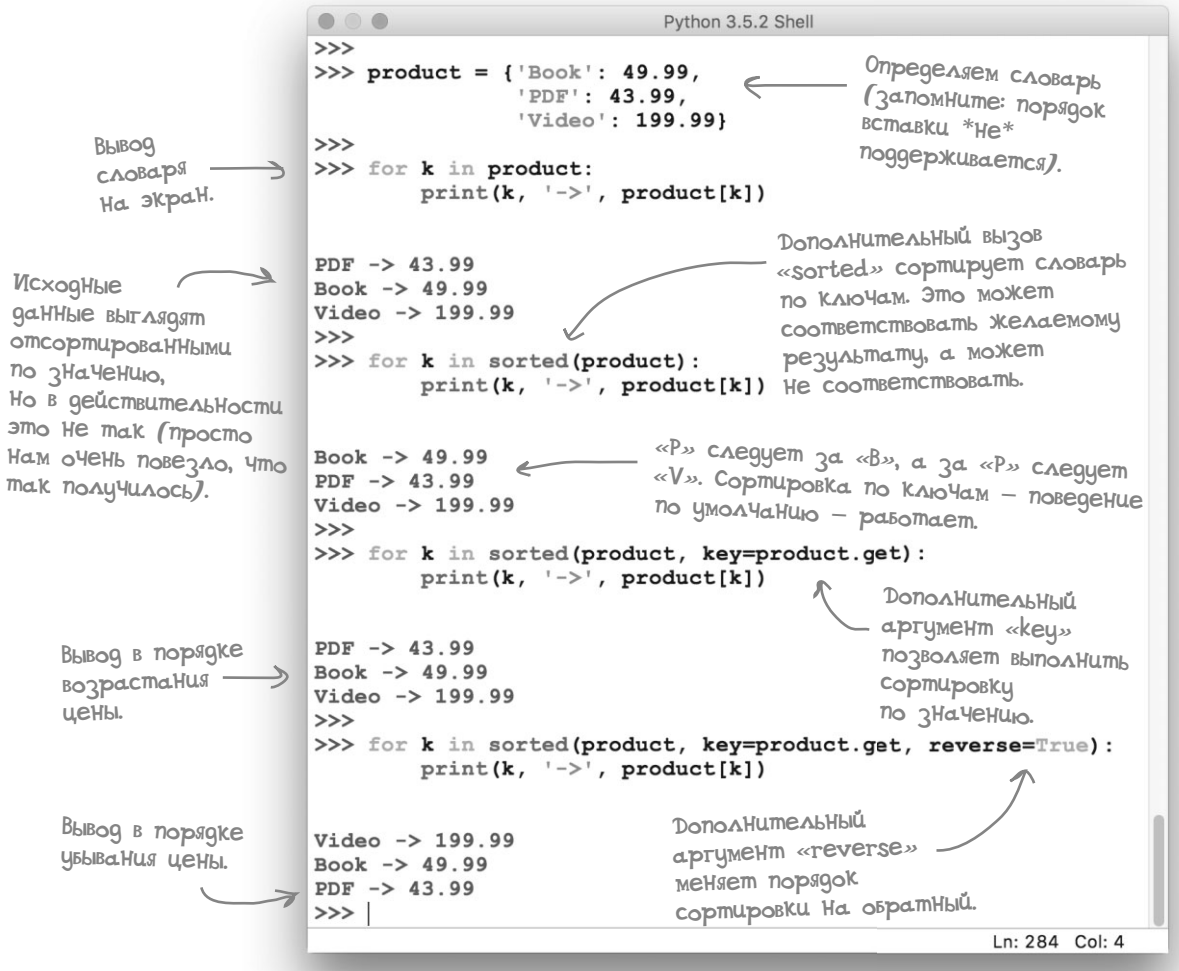
Выбор приема во многом определяется личными предпочтениями, тем не менее третий способ — использование метода `format` — выгодно отличается от двух других (смотрите *PEP 3101* на *https://www.python.org/dev/peps/pep-3101/*). В дикой природе вам обязательно встретится код, использующий тот или иной прием, а иногда (и не всегда к месту) все три. Чтобы узнать больше, начните здесь:

https://docs.python.org/3/library/string.html#formatspec

5. Сортировка

Python обладает замечательными средствами сортировки. Некоторые из встроенных структур данных (например, списки) имеют метод `sort`, который можно использовать для упорядочения данных. Однако по-настоящему неподражаемым Python делает встроенная функция `sorted` (так как она работает с *любыми* встроенными структурами данных).

В сеансе IDLE ниже мы сначала определяем небольшой словарь (`product`), который затем обрабатываем последовательностью циклов `for`. Функция `sorted` управляет порядком, в котором каждый цикл `for` получает данные из словаря. Выполните то же самое на своем компьютере, читая аннотации.



Чтобы узнать больше, как сортировать данные в Python, прочитайте это замечательное руководство:

https://docs.python.org/3/howto/sorting.html#sortinghowto

6. Больше из стандартной библиотеки

Стандартная библиотека Python полным-полна полезных штук. Не пожалейте 20 минут, чтобы посмотреть, что вам доступно, и начните отсюда:

https://docs.python.org/3/library/index.html

Если то, что вам необходимо, уже есть в *стандартной библиотеке*, не тратьте драгоценное время на переписывание. Используйте (и/или расширяйте) то, что доступно. В дополнение к документации Python *Даг Хэллман* (*Doug Hellmann*) адаптировал серию своих статей *Модуль недели* для Python 3, ее можно посмотреть по адресу:

https://pymotw.com/3/

Ниже приведен обзор нескольких наших излюбленных модулей из *стандартной библиотеки*. Мы хотели бы особо подчеркнуть, как важно знать содержимое стандартной библиотеки и что могут дать вам имеющиеся модули.

collections

Этот модуль реализует структуры данных в дополнение к встроенным спискам, кортежам, словарям и множествам. В нем вы найдете много интересного. Вот краткий список того, что есть в `collections`.

- `OrderedDict`: словарь, сохраняющий порядок вставки элементов.
- `Counter`: класс, с помощью которого легко можно посчитать, что угодно.
- `ChainMap`: позволяет представить группу словарей как одно целое.

itertools

Вы уже знаете, насколько хорош цикл `for` в Python, а после переделки в генератор он вообще превращается в безумно крутой инструмент. Модуль `itertools` включает обширный набор инструментов для реализации итераций, но, изучая его богатства, обязательно обратите внимание на классы `product`, `permutations` и `combinations` (а потом сядьте и поблагодарите судьбу за то, что избавила вас от необходимости писать этот код).

functools

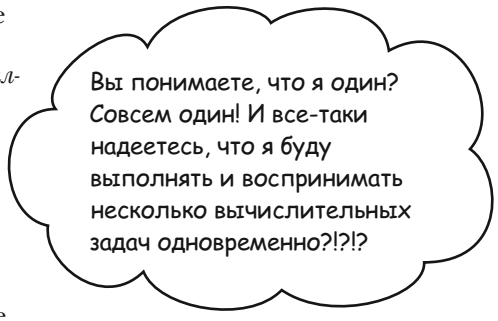
Библиотека `functools` предоставляет широкий ассортимент функций высшего порядка (которые принимают объекты функций в аргументах). Нам больше всего нравится функция `partial`, которая позволяет «замораживать» значения аргументов существующей функции, а затем вызвать эту же функцию под другим именем по вашему выбору. Вы даже не знаете, что теряете, пока не попробуете это.

7. Параллельное выполнение кода

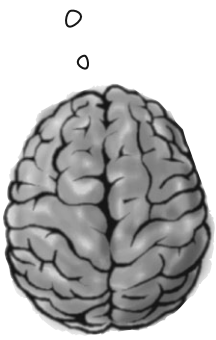
В главе 11¾ мы использовали поток выполнения, чтобы решить проблему ожидания. Потоки — не единственно возможный вариант, когда в программе требуется решать одновременно несколько задач, но будем честными: потоки используются чаще других инструментов и ими нередко злоупотребляют. В этой книге мы намеренно упростили использование потоков, насколько это возможно.

Есть и другие технологии для случаев, когда код должен делать несколько дел сразу. Не в каждой программе возникает такая необходимость, но приятно знать, что когда такая необходимость возникает, Python предлагает множество вариантов в этой области.

Кроме модуля `threading`, в стандартной библиотеке имеется еще несколько модулей, с которыми стоит познакомиться (мы опять отсылаем вас на одну страницу назад, к пункту 6, так как у *Дага Хэллмана* (*Doug Hellmann*) есть несколько замечательных публикаций, посвященных некоторым из этих модулей).



- `multiprocessing`: этот модуль позволяет запускать дополнительные процессы Python и может помочь распределить вычислительную нагрузку между процессорами, если система имеет многоядерный процессор.
- `asyncio`: позволяет организовать параллельное выполнение при помощи создания и определения сопрограмм. Относительно новое дополнение в Python 3, ставшее для многих программистов очень свежей идеей (поэтому единое мнение на этот счет пока не сложилось).
- `concurrent.futures`: позволяет управлять и запускать коллекцию задач одновременно.



Что из этого подходит для вас — это вопрос, на который вы сможете ответить, когда попробуете каждый из модулей в своем коде.

Новые ключевые слова: async и await

Ключевые слова `async` и `await` добавлены в Python 3.5 и представляют стандартный способ создания сопрограмм.

Ключевое слово `async` можно использовать перед ключевыми словами `for`, `with`, и `def` (использованию с `def` уделяется наибольшее внимание). Ключевое слово `await` может использоваться перед (почти) любым другим кодом. В конце 2016 года `async` и `await` были слишком новыми, и программисты на Python только-только начали исследовать их возможности.

Документация Python была обновлена, и теперь в ней есть информация о новых ключевых словах, но более удачное, как нам кажется, описание их использования (и безумия, порождаемого ими) вы найдете в *YouTube*, поискав что-нибудь по этой теме от *Дэвида Бизли* (*David Beazley*). **Имейте в виду**: Дэвид — превосходный рассказчик, но имеет склонность уходить в сторону более продвинутых тем в экосистеме языка Python.

Многие рассказы Дэвида о *GIL* в Python считаются классикой, так же как его книги; больше об этом в *Приложении E*.

Для умников

«GIL» означает «Global Interpreter Lock» (глобальная блокировка интерпретатора). GIL — это внутренний механизм интерпретатора, используемый для обеспечения его стабильности. Его дальнейшее использование в интерпретаторе является предметом многочисленных обсуждений и дискуссий в сообществе Python.

8. Графический интерфейс с использованием Tkinter (и веселье с turtle)

В состав Python входит полноценная библиотека — с именем `tkinter` (*Tk interface*) — для создания кросс-платформенных графических интерфейсов. Вы могли не осознавать этого, но с самой первый главы вы использовали приложение, построенное с помощью `tkinter`, — IDLE.

Библиотека `tkinter` автоматически устанавливается (и готова к использованию), когда вместе с Python устанавливается IDLE (то есть почти всегда). Несмотря на это, `tkinter` не получила широкого использования (и любви), которых заслуживает, так как многие считают ее неуклюжей (по сравнению с некоторыми сторонними альтернативами). Тем не менее, как демонстрирует IDLE, создание полезной и пригодной к употреблению программы с использованием `tkinter` вполне возможно. (Мы упоминали, что `tkinter` устанавливается вместе с Python и готова к использованию?)

Одним из примеров такого использования является модуль `turtle` (также входящий в состав стандартной библиотеки). Цитируя документацию Python: *Черепашья графика – это популярный способ введения в программирование для детей. Она была частью оригинального языка программирования Logo, разработанного Уолли Фойрцойгом (Wally Feurzig) и Сеймуром Пейпертом (Seymour Papert) в 1966 году.* Программисты (в основном дети и забавляющиеся новички) могут использовать команды `left`, `right`, `pendown`, `penup` и другие для рисования на поверхности графического интерфейса (который предоставляет `tkinter`).

Вот небольшая программа — слегка адаптированный вариант примера из документации для модуля `turtle`.

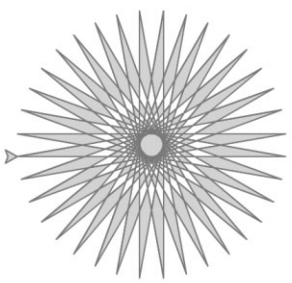
Кроме возможностей «turtle», эта маленькая программа демонстрирует использование цикла «while» и инструкции «break». Они работают в точности как ожидается, но выглядят не так впечатляюще, как цикл «for» и генераторы.

tksample.py - /Users/paul/Documents/tksample.py (3.5.2)

```
from turtle import *

color('purple', 'cyan')
begin_fill()

while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break

end_fill()
done()
```

Ln: 13 Col: 0

Эта маленькая программа использует модуль `turtle` и рисует (на экране) вот такую красивую штуковину.



Мы знаем, что вы можете лучше, но давайте дадим «turtle» шанс?

9. Работа не завершена, пока не проверена

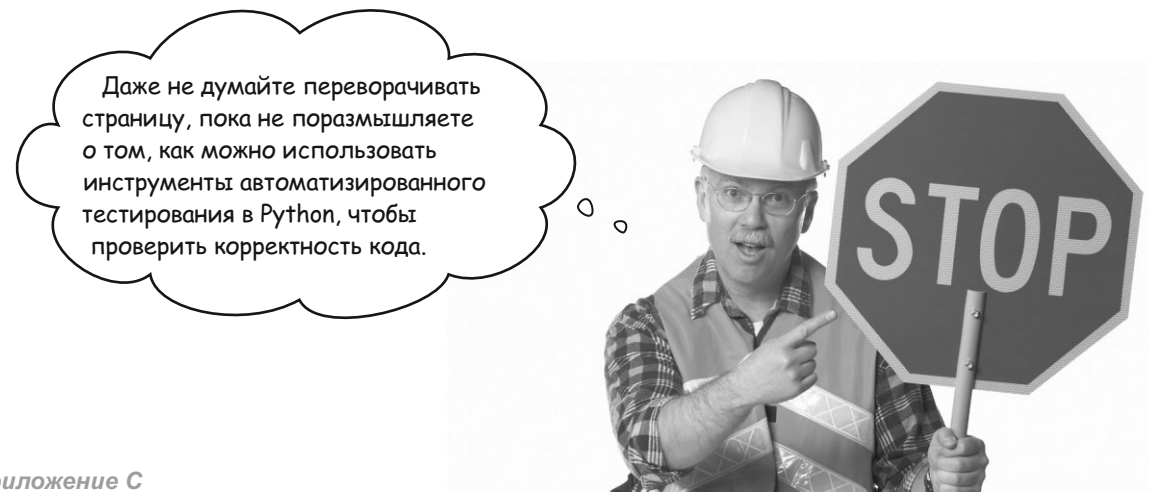
В книге ничего не говорится об автоматизированном тестировании, разве что (в конце главы 4) упоминается инструмент `py.test` для проверки на соответствие *PEP 8*. Это не потому, что мы не считаем автоматизированное тестирование важным. **Напротив, мы считаем его очень важным.** Настолько важным, что этой теме посвящены отдельные книги.

Однако в этой книге мы нарочно не упоминали об инструментах автоматизированной проверки. Это не имеет ничего общего с тем, что мы думаем об автоматизированном тестировании (оно *действительно* очень важно). Однако когда вы начинаете изучать программирование на новом языке, знакомство с приемами автоматизированного тестирования скорее запутывает, чем проясняет, так как создание тестов предполагает хорошее понимание предмета тестирования. А если этот «предмет» является языком программирования, который вы только начинаете изучать... понимаете, куда мы клоним? Что-то вроде проблемы курицы и яйца. Что изучать раньше: код или приемы проверки кода, который вы изучаете?

Конечно, как добросовестный программист на Python вы можете выделить некоторое время, чтобы понять, как *стандартная библиотека* Python упрощает автоматизированное тестирование. Есть два модуля, на которые следует обратить внимание (и изучить).

- `doctest`: этот модуль позволяет встраивать тесты в строки документирования модулей, которые, как бы это странно ни звучало, весьма полезны.
- `unittest`: вы, может быть, уже использовали библиотеку «модульного тестирования» с другими языками программирования, для Python также имеется версия этой библиотеки (которая работает в точности как ожидается).

Те, кто использует модуль `doctest`, обожают его. Модуль `unittest` работает так же, как большинство других библиотек «модульного тестирования» в других языках, и многие программисты на Python жалуются, что он недостаточно *идиоматичен*. Это привело к созданию чрезвычайно популярного `py.test` (и мы поговорим он нем в следующем приложении).



10. Отладка, отладка, отладка

Вам простительно думать, что подавляющее большинство программистов на Python добавляют вызовы `print` в код, когда что-нибудь идет неправильно. И вы будете правы. Это популярный прием отладки.

Другой распространенный прием — экспериментирование в командной строке >>>, который — если задуматься — напоминает сеанс отладки, но *без* обычных отладочных хлопот, таких как трассировка и установка точек останова. Невозможно количественно оценить, насколько командная строка >>> повышает продуктивность программистов на Python. Но мы знаем, что если в будущем выпуске Python удалят интерактивную оболочку, это станет большой неприятностью.

Узнать все о трассировке и точках останова можно в документации к «pdb».

Если у вас есть код, который не делает то, что он должен делать, а вызовы `print` и эксперименты в командной строке >>> ничего не дают, используйте встроенный в Python отладчик `pdb`.

Отладчик `pdb` можно запустить прямо в окне терминала операционной системы, используя такую команду (где `myprog.py` — это программа, которую нужно исправить):

```
python3 -m pdb myprog.py
```

Как всегда, в Windows нужно использовать «py -3» вместо «python3» (то есть «py», пробел, а затем минус 3).

С отладчиком `pdb` можно взаимодействовать из командной строки >>>, и это воплощение «лучшего из обоих миров», к которому, мы надеемся, вы когда-нибудь придете. Особенности работы и все обычные команды отладчика (установка точек останова, пропуск функций, вход в функции и так далее) рассмотрены в документации:

https://docs.python.org/3/library/pdb.html

Технология `pdb` не является чем-то второстепенным или «для галочки»; это великолепный полнофункциональный отладчик для Python (и он входит в состав дистрибутива).

Постарайтесь освоить отладчик «pdb», чтобы сделать его частью своего инструментария.

Приложение D: топ-10 проектов, которые мы не рассмотрели

Еще больше инструментов, библиотек и модулей

Мы знаем, о чем вы подумали, читая заголовок этого Приложения.

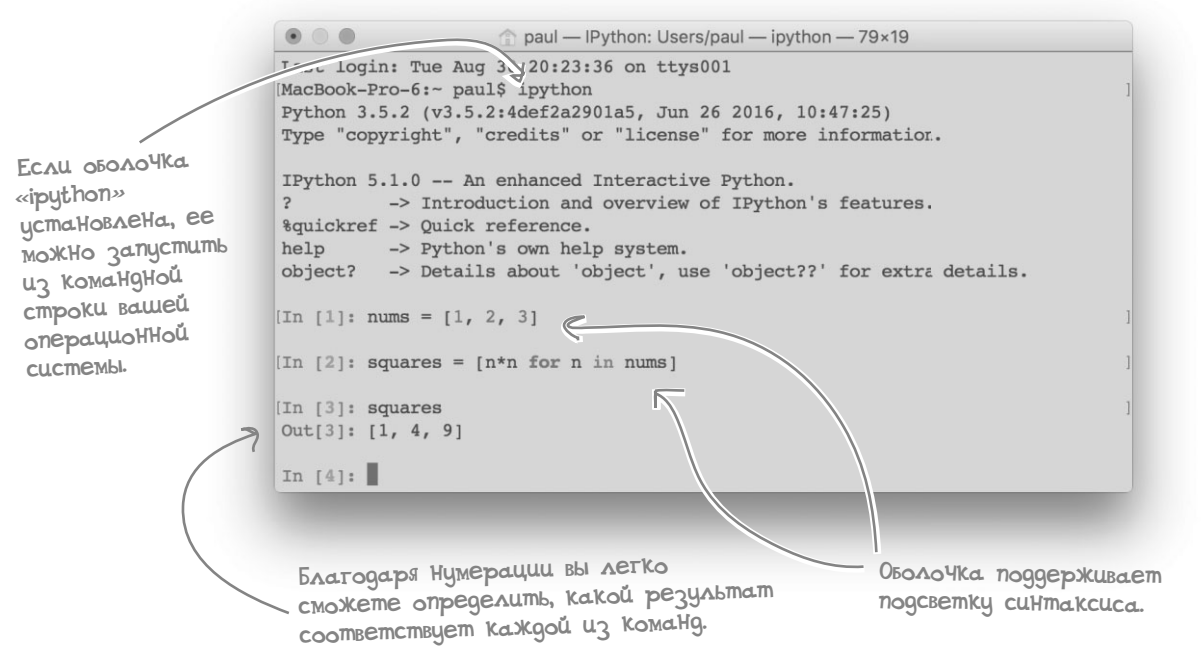
Почему они не назвали предыдущее приложение: *«Топ-20 тем, которые не были рассмотрены»*? Зачем нам еще 10? В предыдущем приложении мы ограничили обсуждение инструментами, которые поставляются вместе с Python (те самые «батарейки», которые входят в комплект). В этом приложении мы пойдем дальше и обсудим группу технологий, которые стали доступны, *потому что* существует Python. Здесь перечислено много полезного, но, как в случае с предыдущим приложением, если вы бегло просмотрите оставшиеся страницы, то не пропустите *ничего важного*.

1. Альтернативы командной строке >>>

На протяжении книги мы использовали встроенную интерактивную оболочку Python >>>, работая с ней в окне терминала или в IDLE. Поступая так, мы хотели показать, насколько полезной может быть командная строка >>> для экспериментов с различными идеями, при изучении библиотек и тестировании кода.

Существует много альтернатив встроенной командной строке >>>, но одна из них — `ipython` — заслуживает большего внимания. Если вы хотите получить больше возможностей, чем дает простая командная строка >>>, присмотритесь к `ipython`. Эта командная оболочка очень популярна среди программистов на Python, но *особой* популярностью она пользуется в научном сообществе.

Чтобы получить представление о возможностях `ipython`, в сравнении с обычной командной строкой >>>, посмотрите на следующий короткий интерактивный сеанс работы в `ipython`.



Узнайте больше о `ipython` на *https://ipython.org*.

Существуют также другие альтернативы командной строке >>>, но только одна из них (на наш взгляд) сопоставима с `ipython`. И это `ptpython` (подробности на *https://pypi.org/project/ptpython/*). Если вам нравится работать с текстовым терминалом, но вы хотите найти что-то более «полноэкранное» (в отличие от `ipython`), возьмите `ptpython`. Он вас не разочарует.

Тише! С тех пор как Пол открыл для себя «ptpython», он пользуется им каждый день.

Оболочки «ipython» и «ptpython» можно установить с помощью «pop», как любые другие сторонние модули.

2. Альтернативы интегрированной среде IDLE

Мы смело заявляем, что испытываем слабость к IDLE. Нам очень нравится, что в состав Python входит не только мощная интерактивная оболочка >>>, но и довольно неплохой редактор с графическим интерфейсом и отладчиком. Немногие из популярных языков программирования могут похвастать чем-то подобным в стандартном дистрибутиве.

К сожалению, среда *IDLE* не слишком популярна среди разработчиков на Python, потому что кажется бедной по сравнению с возможностями «профессиональных» предложений. Мы думаем, это несправедливое сравнение, потому что IDLE и не разрабатывалась для соревнований в этом сегменте. Главная ее задача — позволить новым пользователям начать работу и продвигаться быстро, и с этой задачей она прекрасно справляется. Поэтому мы считаем, что IDLE заслуживает более уважительного отношения в сообществе Python.

Оставьте IDLE в стороне, если потребуется более профессиональная IDE, — вам есть из чего выбирать. Наиболее популярными в мире Python являются:

- *Eclipse*: ***https://www.eclipse.org***
- *PyCharm*: ***https://www.jetbrains.com/pycharm/***
- *WingWare*: ***https://wingware.com***

Eclipse — технология с полностью открытым исходным кодом, поэтому вам она будет стоить ровно столько, сколько стоит ее загрузка. Если вы уже фанат *Eclipse*, вам будет приятно узнать, что она поддерживает Python. Но если вы еще не используете *Eclipse*, мы не рекомендуем ее, потому что есть *PyCharm* и *WingWare*.

PyCharm и *WingWare* — коммерческие продукты, имеющие «версии для сообщества» (с несколько ограниченными возможностями), которые можно загрузить бесплатно. В отличие от *Eclipse*, которую можно использовать с различными языками программирования, *PyCharm* и *WingWare* нацелены именно на Python. Как и все IDE, они имеют великолепную поддержку проектов, систем управления версиями (например, *git*) и коллективной разработки, открывают доступ к онлайн-документации Python и т. д. Мы советуем вам попробовать обе и лишь потом выбирать одну.

Если все эти IDE не для вас, не огорчайтесь: все самые знаменитые текстовые редакторы поддерживают Python.

Что использует Пол?

Пол выбирает `vim` (он пользуется *MacVim* на всех своих компьютерах). Работая с проектами на Python, Пол совмещает использование `vim` с `ptpython` (когда экспериментирует с кодом), а также он большой поклонник IDLE. Для управления версиями на локальной машине Пол использует `git`.

Там, где не нужно, Пол не использует полнофункциональную IDE, а его студенты предпочитают *PyCharm*. Пол также использует (и советует) интегрированную среду *Jupyter Notebook*, которую мы обсудим далее.

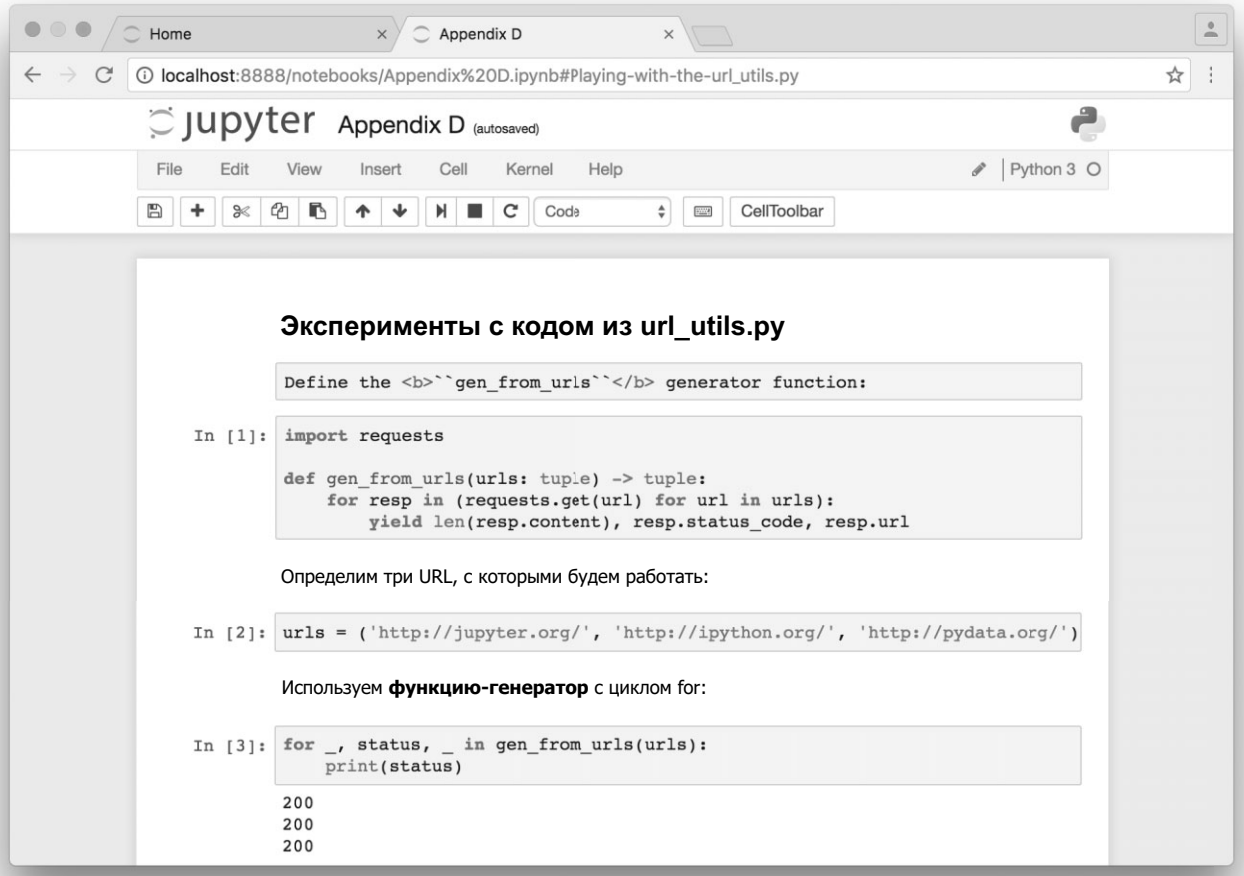
3. Jupyter Notebook: IDE на основе веб

В пункте 1 мы обратили ваше внимание на оболочку `ipython` (которая является прекрасной альтернативой >>>). Та же команда разработчиков выпустила интегрированную среду *Jupyter Notebook* (ранее известную как *iPython Notebook*).

Новое поколение Jupyter Notebook называется Jupyter Lab и во время работы над этой книгой находилось в состоянии «альфа-версии». Следите за проектом Jupyter Lab: это будет что-то действительно особенное.

Jupyter Notebook можно охарактеризовать как мощь `ipython`, заключенную в интерактивную веб-страницу (с обобщенным названием «notebook», то есть «блокнот»). Самое примечательное в *Jupyter Notebook* — возможность править и запускать код, не покидая страницу, и — если есть необходимость — также можно добавлять текст и графические элементы.

Вот код из главы 12, открытый в *Jupyter Notebook*. Обратите внимание, как мы добавили текстовые описания в блокнот, чтобы объяснить, что происходит.



Узнать больше о *Jupyter Notebook* можно на сайте *http://jupyter.org*, а для установки можно использовать утилиту `pip`. Вы не разочаруетесь. *Jupyter Notebook* — потрясающее приложение на Python.

4. Наука работы с данными

Говоря о практическом применении Python, нельзя не упомянуть одну из областей, которая продолжает интенсивно развиваться: **науку о данных**.

И это не случайно. Инструменты, доступные специалистам по работе с данными, которые используют Python, — это инструменты мирового уровня (на зависть многим другим сообществам программистов). Особенно хорошо для специалистов, не связанных с наукой о данных, то, что инструменты, популярные в этих специфичных кругах, находят широкое применение за пределами обработки *больших данных*.

Использованию Python в *науке о данных* посвящены (и продолжают посвящаться) целые книги.

Вот подборка библиотек и модулей для работы с данными (или любых других научных вычислений). Если *наука о данных* — не ваша тема, все равно просмотрите список — здесь много интересного.

- `bokeh`: множество технологий для создания интерактивных графиков на веб-страницах.
- `matplotlib/seaborn`: исчерпывающее множество модулей для отображения графиков (интегрированых также в `ipython` и *Jupyter Notebook*).
- `numpy`: кроме всего прочего, позволяет эффективно хранить и обрабатывать многомерные данные. Если вы фанат матриц, вы полюбите `numpy`.
- `scipy`: множество научных модулей, оптимизированных для анализа численных данных, которые расширяют и дополняют то, что поставляется с `numpy`.
- `pandas`: пришедшие в Python из *R* будут чувствовать себя с пакетом `pandas` совсем как дома, потому что `pandas` предоставляет структуры данных и инструменты для оптимизированного анализа (и построен на основе `numpy` и `matplotlib`). Необходимость использования `pandas` привела (и еще долго будет приводить) в сообщество специалистов по работе с данными. `pandas` — еще одно *потрясающее* применение Python.
- `scikit-learn`: множество алгоритмов машинного обучения и технологий, реализованных на Python.

Обратите внимание: большинство из этих библиотек можно установить с помощью `pip`.

Исследовать область пересечения Python и *науки о данных* лучше всего с веб-сайта PyData: http://pydata.org. Щелкните по ссылке *Downloads* (Загрузки), и вы удивитесь изобилию доступных продуктов (все с отрытым исходным кодом). Наслаждайтесь!

5. Технологии веб-разработки

Python занимает прочное место в веб-пространстве, но *Flask* (вместе с *Jinja2*) — не единственный фреймворк для создания серверной части веб-приложений (даже притом что *Flask* очень популярен, особенно если у вас скромные потребности).

Наиболее известная технология создания веб-приложений на Python — это *Django*. Мы его не использовали, потому что (в отличие от *Flask*) для создания первого веб-приложения с *Django* нужно узнать и понять немного больше (*Django* выходит за пределы книги, в которой изучаются основы Python). Главная причина, почему *Django* так популярен среди программистов Python, — он правда очень, очень хорош.

Если вы причисляете себя к веб-разработчикам, потратьте время и хотя бы пролистайте руководство по *Django*. Сделав это, вы будете лучше понимать, стоит ли оставаться с *Flask* или перейти на *Django*.

Если вы перейдете на *Django*, то окажетесь в очень хорошей компании. Вокруг *Django* сплотилось большое сообщество (внутри еще более обширного сообщества Python), которое даже имеет собственную конференцию *DjangoCon*. К настоящему времени прошло уже несколько конференций *DjangoCon* в США, Европе и Австралии. Вот несколько ссылок, по которым вы сможете найти дополнительные сведения:

- домашняя страница Django (где есть ссылка на руководство пользователя):
  https://www.djangoproject.com
- DjangoCon в США:
  https://djangocon.us
- DjangoCon в Европе:
  https://djangocon.eu
- DjangoCon в Австралии:
  http://djangocon.com.au

Но это еще не все

Кроме *Flask* и *Django*, есть другие веб-фреймворки (и мы знаем, что не упомянем чьи-то любимые). Вот те, о которых мы слышали: *Pyramid*, *TurboGears*, *web2py*, *CherryPy* и *Bottle*. Более полный список можно найти на вики-странице Python:

https://wiki.python.org/moin/WebFrameworks

6. Работа с данными в веб

В главе 12 мы воспользовались библиотекой `requests`, чтобы показать, как здорово работает наш генератор (по сравнению с эквивалентным ему генератором списков). Наше решение использовать `requests` не было случайным. Если вы спросите, какой модуль для работы с веб лучше, большинство разработчиков на Python ответят «requests».

PyPI: Python Package Index (каталог пакетов для Python) обитает на https://pypi.org/

Модуль `requests` позволяет работать с HTTP и веб-службами через простой, но мощный прикладной интерфейс Python. Даже если ваша ежедневная работа не связана с веб, вы многое узнаете, просто просмотрев код `requests` (весь проект `requests` можно рассматривать как мастер-класс разработки на Python).

Узнать больше о `requests` можно здесь:

http://docs.python-requests.org/en/master/

Под скрапингом вообще подразумевается загрузка и анализ данных. Под веб-скрапингом — загрузка и анализ содержимого веб-страниц. — Прим. науч. ред.

Скрапинг веб-данных

Поскольку в своей основе веб — это текстовая платформа, Python всегда прекрасно работал в этом пространстве: *стандартная библиотека* включает модули для работы с JSON, HTML, XML и другими похожими текстовыми форматами, а также с соответствующими протоколами Интернета. В следующих разделах документации Python можно найти список модулей, которые входят в стандартную библиотеку и представляют наибольший интерес для веб/интернет-программистов:

- Работа с данными в Интернет:
  https://docs.python.org/3/library/netdata.html
- Инструменты для обработки структурированной разметки:
  https://docs.python.org/3/library/markup.html
- Протоколы Интернет и поддержка:
  https://docs.python.org/3/library/internet.html

Если потребуется работать с данными, доступными только через статические веб-страницы, вы, скорее всего, захотите извлечь эти данные (быстрое введение в скрапинг доступно по адресу *https://en.wikipedia.org/wiki/Web_scraping*). Для Python доступны сторонние модули, которые сэкономят массу времени.

- *Beautiful Soup:*
  https://www.crummy.com/software/BeautifulSoup/
- *Scrapy:*
  http://scrapy.org

Попробуйте обе библиотеки, чтобы определить, какая больше подходит для решения ваших задач, а затем установите подходящую.

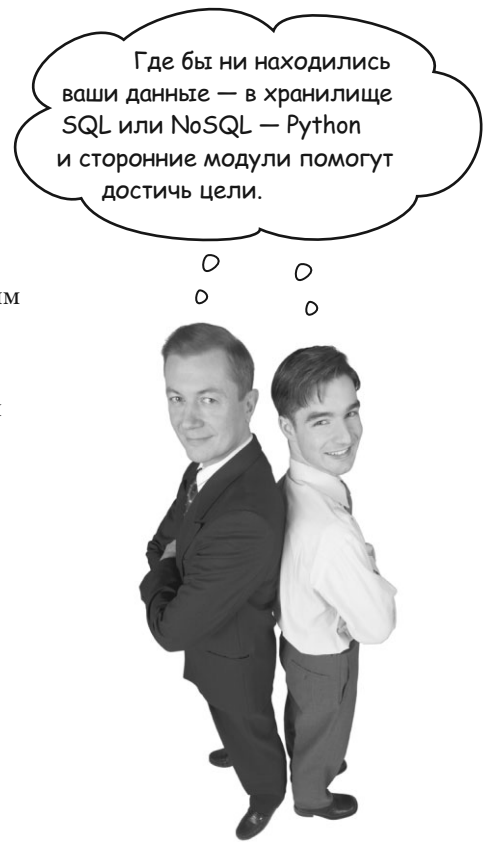
7. Еще больше источников данных

Чтобы быть ближе к реальности (и как можно более проще), мы использовали базу данных *MySQL*. Если вы уже давно работаете с языком SQL (независимо от конкретной базы данных), отвлекитесь немного от своих дел и потратьте пару минут, чтобы с помощью `pip` установить `sqlalchemy` — возможно, это будет ваше *лучшее* решение.

Модуль `sqlalchemy` предназначен для знатоков SQL так же, как `requests` — для знатоков веб. Он необходим. Проект *SQL Alchemy* предоставляет множество высокоуровневых технологий на основе Python для работы с табличными данными (хранящимися в базах данных *MySQL*, *PostgreSQL*, *Oracle*, *SQL Server* и т. д.). Если вам понравилось то, что мы сделали с модулем *DBcm*, вы полюбите модуль *SQL Alchemy*, который позиционируется как набор инструментов для работы с базами данных из программ на Python.

Узнайте больше об этом проекте по адресу:

http://www.sqlalchemy.org



Другие запросы, кроме SQL

Не все данные, которые могут пригодиться, нужно хранить в базах данных SQL, поэтому могут настать времена, когда *SQL Alchemy* не поможет. Базы данных NoSQL считаются хорошим дополнением для любого вычислительного центра, а классическим примером и наиболее популярным выбором считается *MongoDB* (хотя вариантов много).

Если вы работаете с данными в формате JSON или в нетабличном (но структурированном) виде, *MongoDB* (или что-то подобное) может оказаться именно тем, что вам нужно. Узнайте больше о *MongoDB* здесь:

https://www.mongodb.com

Также посмотрите, как Python поддерживает программирование *MongoDB* с использованием драйвера базы данных `pymongo`, на странице с документацией *PyMongo*:

https://api.mongodb.com/python/current/

8. Инструменты для программирования

Вы можете считать свой код замечательным, но ошибки все же случаются.

В таких случаях Python может предложить вам в помощь: командную строку >>>, отладчик `pdb`, *IDLE*, функции `print`, `unittest` и `doctest`. Если всего этого недостаточно, есть сторонние модули, которые могут помочь.

Возможно, вы совершите классическую ошибку, которую другие совершали много раз. Или забудете импортировать нужные модули и заметите проблему только тогда, когда начнете демонстрировать, как отлично работает ваш код, в комнате, полной посторонних людей (упс).

Чтобы избежать подобных неприятностей, установите *PyLint*, инструмент анализа кода на Python:

https://www.pylint.org

PyLint проанализирует код и сообщит о возможных проблемах *прежде*, чем вы его запустите в первый раз.

Если вы проверите свой код с помощью *PyLint* прежде, чем запустить его в комнате, полной посторонних, вы почти наверняка предотвратите неловкий момент. *PyLint* может, конечно, ранить ваши чувства, потому что никто не любит, когда ему говорят, что его код неидеален. Но эта боль оправдана (а может, и полезна: *ущемленное самолюбие лучше, чем публичный конфуз*).

Больше помощи с тестированием

В *Приложении C* (пункт 9) мы обсудили встроенную поддержку автоматизированного тестирования в Python. Есть и другие подобные инструменты, вы уже знаете, что `py.test` — один из них (потому что мы использовали его раньше для проверки на совместимость с *PEP 8*).

Фреймворки для тестирования похожи на веб-фреймворки: у каждого есть свой любимый. К слову сказать, многие программисты на Python не очень уважают `py.test`, поэтому мы советуем вам познакомиться с ним поближе:

http://doc.pytest.org/en/latest/

9. Kivy: наш выбор в номинации «Лучший проект всех времен»

В мире мобильных сенсорных устройств Python не так силен, как хотелось бы. Тому есть много причин (но мы не будем на них останавливаться). Достаточно сказать, что на момент публикации книги было довольно сложно создать приложение для *Android* или *iOS* на чистом Python.

Единственный проект, в котором наметился прогресс в этой области, — это *Kivy*.

Kivy — библиотека для Python, которая позволяет разрабатывать приложения, использующие мультисенсорные интерфейсы. Посмотрите на официальной странице *Kivy*, что предлагает библиотека:

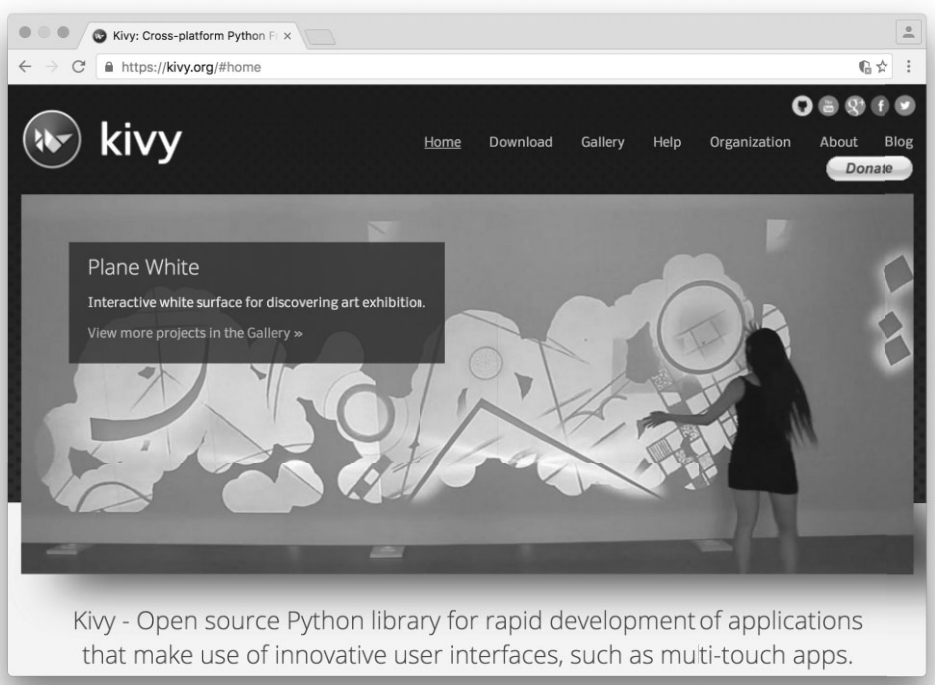
https://kivy.org

На этой странице щелкните по ссылке *Gallery* (Галерея) и спокойно дождитесь, когда страница загрузится. Если какой-то проект в галерее привлек ваше внимание, щелкните на графическом элементе, чтобы получить дополнительную информацию и посмотреть демонстрационный пример. Рассматривая пример, помните: *все, что вы видите, написано на Python*. По ссылке *Blog* (Блог) можно получить дополнительную информацию.

Самое замечательное, что код пользовательского интерфейса *Kivy* был написан один раз и на всех поддерживаемых платформах используется *без изменений*.

Если вы хотите внести посильную лепту в развитие какого-нибудь проекта на Python, подумайте, не пришло ли время помочь *Kivy*: это замечательный проект, над ним работает отличная команда, которая решает технически сложную проблему. По крайней мере, скучать не придется.

Скриншот страницы Kivy, сделанный в 2016-м, демонстрирует одну из реализаций: сенсорный интерфейс для полного погружения.



Kivy — библиотека Python с открытым исходным кодом для быстрой разработки приложений, в которых используются инновационные пользовательские интерфейсы, в том числе мультисенсорные.

10. Альтернативные реализации

Из пункта 1 *Приложения C* вы знаете, что есть несколько реализаций языка Python (Python 2 и Python 3). Это означает, что существует по крайней мере два интерпретатора Python: один выполняет код Python 2, а другой — Python 3 (который мы использовали в книге). Интерпретаторы Python, доступные для загрузки на официальном сайте Python (как описывалось в *Приложении A*), называют также *эталонными реализациями CPython*. *CPython* — это версия Python, распространяемая разработчиками Python, получившая такое имя потому, что написана на переносимом C: она спроектирована так, чтобы ее легко было переносить на другие платформы. Как рассказывалось в *Приложении A*, есть возможность загрузить дистрибутивы для *Windows* и *Mac OS X*, а кроме того, интерпретатор может быть уже установлен в вашем дистрибутиве *Linux*. Все эти интерпретаторы основаны на *CPython*.

Python — проект с открытым исходным кодом, каждый может взять *CPython* и изменить его, как заблагорассудится. Разработчики также могут реализовать собственный интерпретатор Python на любом языке программирования, с использованием любых компиляторов и на любых платформах. Эта работа не для слабонервных, тем не менее многие занимаются ею (некоторые просто ради забавы). Вот короткие описания и ссылки на несколько подобных проектов.

- *PyPy* (произносится как «пай-пай») — экспериментальный компилятор для Python 2 (хотя в будущем будет предусмотрена поддержка Python 3). PyPy принимает код на Python и запускает процесс компиляции, в результате которого получается продукт, работающий иногда быстрее, чем *CPython*. Узнать больше можно по ссылке:

  http://pypy.org

- *IronPython* — версия Python 2 для платформы *.NET*:

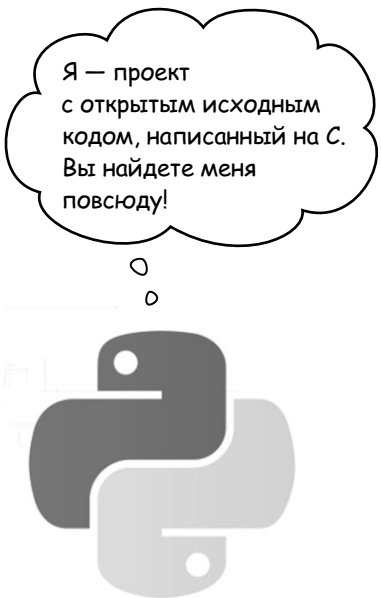
  http://ironpython.net

- *Jython* — версия Python 2, которая работает в виртуальной машине Java:

  http://www.jython.org

- *MicroPython* — версия Python 3 для использования на микроконтроллере *pyboard*, который не больше спичечного коробка. Возможно, это самое замечательное миниатюрное устройство, которое вы когда-либо видели. Посмотрите:

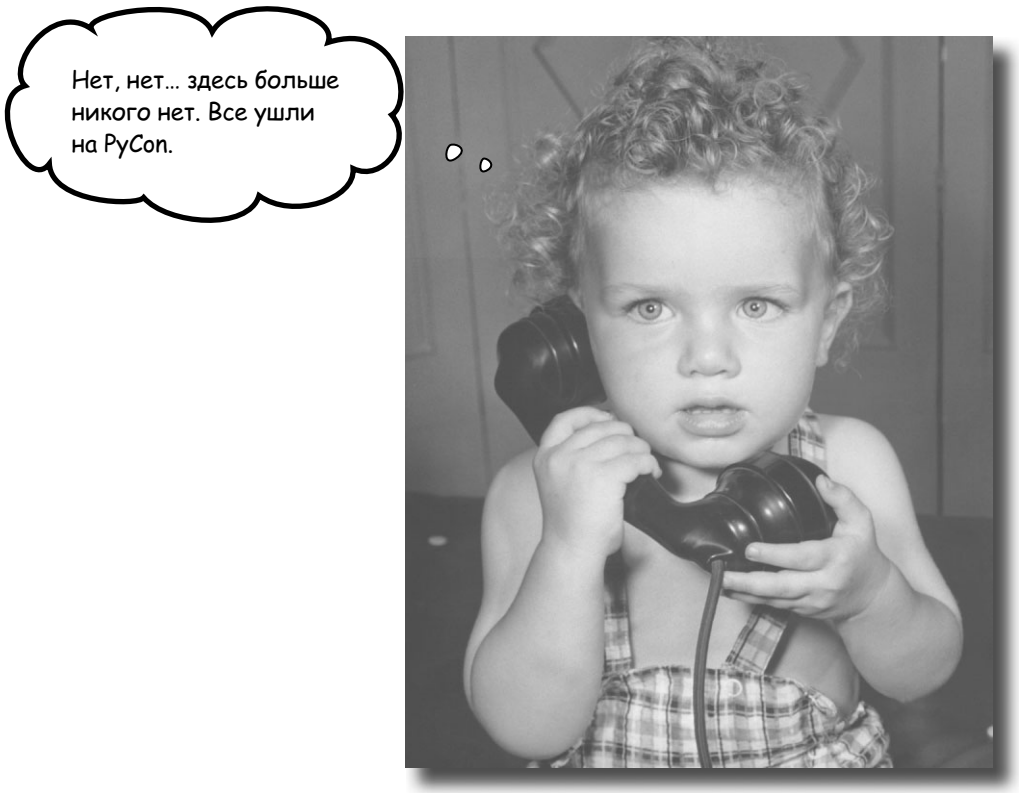
  http://micropython.org



- Несмотря на существование альтернативных интерпретаторов Python, большинство программистов довольствуются *CPython*. Все больше разработчиков предпочитают Python 3.

Приложение E: присоединяйтесь

Сообщество Python



Python — это гораздо больше, чем отличный язык программирования.

Это отличная компания. Сообщество Python гостеприимное, разнообразное, открытое, дружественное, щедрое и готовое делиться. Просто удивительно, что еще никто не напечатал такую поздравительную открытку! Если серьезно, то Python — не столько язык, сколько особая наука программирования. Вокруг Python сформировалась целая экосистема: отличные книги, блоги, веб-сайты, конференции, встречи, группы пользователей и великие личности. В этом приложении мы познакомимся с сообществом Python и посмотрим, что оно может предложить. Не пишите код в одиночестве: **присоединяйтесь к сообществу!**

BDFL: Benevolent Dictator for Life — великодушный пожизненный диктатор

Гвидо ван Россум (*Guido van Rossum*) — голландский программист, подаривший миру язык программирования Python (который начинался как хобби в конце 1980-х). Продолжение разработки и направление развития языка определяют *основные разработчики Python*, и Гвидо — один из них (хотя и очень важный). Титул Гвидо — Великодушный пожизненный диктатор — является признанием центральной роли, которую он продолжает играть в жизни Python. Увидев аббревиатуру BDFL в связи с Python, знайте, что под ней подразумевается Гвидо.

Гвидо официально утверждает, что название «Python» — это кивок (и подмигивание) в сторону Британского телевизионного комедийного сериала *Monty Python's Flying Circus* (Летающий Цирк Монти Пайтона). Этим объясняется использование имени `spam` для многих переменных, упомянутых в документации Python.

Несмотря на ведущую роль, Гвидо **не** владеет Python: им вообще никто не владеет. Однако интересы языка защищены PSF.

PSF: The Python Software Foundation — Фонд содействия развитию Python

PSF — это некоммерческая организация, которая заботится об интересах Python. Управляет ею назначаемый/избираемый совет директоров. PSF способствует разработке языка и спонсирует ее продолжение. Вот цитата из определения целей PSF:

> *Цель Фонда содействия развитию Python – способствовать, защищать и продвигать язык программирования Python, поддерживать и содействовать росту разнообразного и интернационального сообщества программистов на Python.*

Выскажите свое мнение: присоединитесь к PSF.

Кто угодно может присоединяться к фонду PSF и принять участие в его работе. Более подробную информацию можно найти на сайте PSF:

https://www.python.org/psf/

Одним из главных направлений в деятельности PSF является участие (и организация) *PyCon* — ежегодной конференции, посвященной Python.

Присоединяйтесь: посещайте PyCon.

PyCon: конференция, посвященная Python

Кто угодно может присутствовать (и выступить) на конференции PyCon. В 2016 году в Портленде (Portland), штат Орегон (Oregon), прошла конференция, которую посетили тысячи разработчиков на Python (предыдущие две конференции PyCon проходили в канадском Монреале). PyCon является самой большой конференцией по Python, хотя и не единственной. Конференции по Python проводятся по всему миру: от самых маленьких, региональных (из десятков участников) до средних, национальных (с сотнями участников) и подобных *EuroPython* (с тысячами участников).

Чтобы узнать, проводится ли PyCon поблизости от вас, поищите слово «PyCon» вместе с названием ближайшего к вам города (или страны, где вы живете). Возможно, результаты поиска вас приятно удивят. Посещение местных PyCon — это отличный способ встретиться и пообщаться с единомышленниками-разработчиками. Многие выступления и заседания на различных конференциях PyCon есть в записи: зайдите на *YouTube* и поищите по слову «PyCon», чтобы увидеть, что доступно для просмотра.

Толерантное сообщество: уважение к многообразию

Из всех конференций по программированию, существующих сегодня, PyCon стала одной из первых, где был введен и соблюдается *Кодекс поведения*. Ознакомиться с Кодексом поведения на PyCon 2016 можно здесь:

https://us.pycon.org/2016/about/code-of-conduct/

Кодекс поведения на русском языке для PyCon Russia 2017 можно найти по адресу: http://pycon.ru/2017/conference/coc/

Это *очень правильное решение*. Многие и многие небольшие региональные PyCon принимают Кодекс поведения, что также очень приветствуется. Сообщество становится сильным и всеохватывающим, когда есть четкое понимание того, что приемлемо, а что нет, и Кодекс поведения помогает поддерживать дружелюбие и гостеприимность всех конференций PyCon во всем мире.

Кроме стремления к всеобщему доброжелательному отношению, было увеличено представительство конкретных групп внутри сообщества Python, особенно тех, которые — традиционно — представлены недостаточно полно. Среди них наиболее известна группа *PyLadies*, созданная, чтобы помочь «большему количеству женщин стать активными участниками и лидерами в открытом сообществе Python». Если вам повезет, то поблизости окажется «отделение» *PyLadies*. Можно это выяснить, начав поиски с веб-сайта *PyLadies*:

http://www.pyladies.com

Как и сообщество Python, движение *PyLadies* начиналось с малого, но очень быстро выросло до глобального масштаба (что поистине вдохновляет).

Приходят ради языка, остаются ради сообщества

Новые программисты на Python отмечают, насколько многообразно сообщество Python. Во многом оно сформировалось благодаря личному примеру и роли Гвидо, осуществляющему руководство твердо, но доброжелательно. Есть и другие ведущие светила, а также много вдохновляющих историй.

Поощряйте и поддерживайте многообразие внутри сообщества Python.

Нет ничего более захватывающего, чем речь *Наоми Чедер* (*Naomi Ceder*) на *EuroPython* (которую повторили другие региональные конференции, в том числе *PyCon Ireland*). Вот ссылка на речь Наоми, которую мы рекомендуем посмотреть:

https://www.youtube.com/watch?v=cCCiA-IlVco

Наоми рассказывает о жизни в Python, о том, как сообщество поддерживает многообразие, говорит, что в нем всегда найдется работа для всех желающих.

Один из способов узнать больше о сообществе — слушать подкасты, создаваемые его участниками. Далее мы обсудим два подкаста, посвященных Python.

Подкасты Python

Сейчас существуют подкасты на *все* случаи жизни. В сообществе Python есть два подкаста, на которые, как нам кажется, стоит подписаться и слушать, например во время поездки, велосипедной прогулки, утренней пробежки или на досуге. Оба подкаста заслуживают вашего внимания:

- *Talk Python to Me*: **https://tlkpython.fm**
- *Podcast.__init__*: **http://pythonpodcst.com**

Следуйте за обоими подкастами в *Twitter*, расскажите друзьям о них и окажите создателям этих подкастов полную поддержку. Оба подкаста, *Talk Python to Me* и *Podcast.__init__*, созданы постоянными членами сообщества Python для нашей общей пользы (*не* для прибыли).

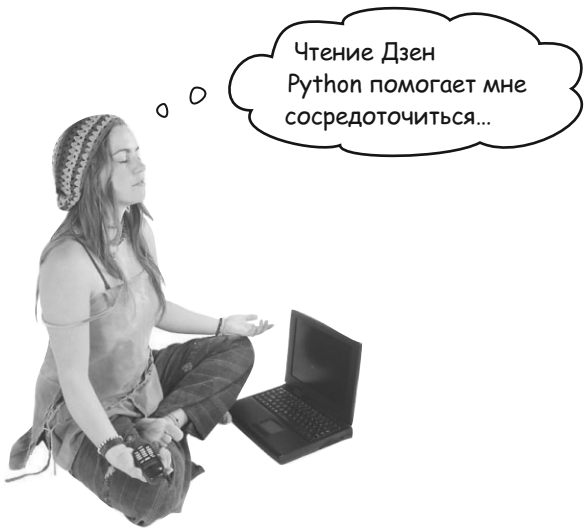
Информационные бюллетени Python

Если подкасты — это не ваше, но вы все еще хотите следить за тем, что происходит в мире Python, есть три еженедельных информационных бюллетеня, которые могут вам пригодиться:

- Pycoder's Weekly: **http://pycoders.com**
- Python Weekly: **http://www.pythonweekly.com**
- Import Python: **http://importpython.com/newsletter**

Специальные информационные бюллетени содержат ссылки на самые разные источники: блоги, видеоблоги, статьи, книги, видео, выступления, новые модули и проекты. Их еженедельные объявления доставляются прямо на ваш электронный адрес. Идите и подписывайтесь.

Помимо фонда, множества конференций, подгрупп, таких как *PyLadies*, кодекса поведения, признания многообразия, подкастов и информационных бюллетеней, Python имеет также свой собственный, особый *Дзен*.



Дзен Python

Много лун назад, Тим Петерс (Tim Peters, один из первых светил в Python) сел и задумался: *что же делает Python тем самым Python?*

Ответ явился Тиму в виде *Дзен Python*, который вы можете прочитать, начиная с любой версии интерпретатора, введя в командной строке >>> заклинание:

```
import this
```

Мы сделаем это за вас и покажем вывод вышеуказанной строки кода на скриншоте внизу страницы. Обязательно читайте *Дзен Python* не реже одного раза в месяц.

Многие пытались сжать *Дзен Python* в какой-нибудь маленький легкий дайджест. Но никто кроме `xkcd` не смог это сделать. Если у вас есть выход в Интернет, введите следующую строку в интерактивной оболочке >>>, чтобы увидеть (буквально) как `xkcd` сделал это:

```
import antigravity
```

Python 3.5.2 Shell

```
>>>
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Ln: 28 Col: 4

Запомните: Дзен нужно читать *не реже* одного раза в месяц.

Дзен Python от Тима Петерса

Красивое лучше уродливого.

Явное лучше неявного.

Простое лучше сложного.

Сложное лучше запутанного.

Плоское лучше вложенного.

Разреженное лучше плотного.

Читаемость имеет значение.

Особые случаи не настолько особые,
чтобы нарушать правила.

При этом практичность важнее безупречности.

Ошибки никогда не должны замалчиваться.

Если не замалчиваются явно.

Встретив двусмысленность, подави искушение угадать.

Должен существовать один — и, желательно, только один —
очевидный способ сделать что-то.

Хотя он поначалу может быть не очевиден, если вы
не голландец.

Сейчас лучше, чем никогда.

Хотя часто никогда лучше, чем *прямо* сейчас.

Если реализацию сложно объяснить — идея плоха.

Если реализацию легко объяснить — идея, возможно, хороша.

Пространства имен — отличная штука! Не экономь на них!

Алфавитный указатель

А

Автоматизированное тестирование 584

Автоматический перезапуск веб-приложений 263

Адреса URL
- и декораторы функций 243, 245, 432, 444
- обработка с помощью генератора списков 545
- ограничение доступа 418, 432, 444

Аннотации (функции) 198

Аргументы
- *args 426, 437
- **kwargs 428, 437
- self 353, 355, 358
- в словарях 428
- декораторов функций 259, 426
- добавление нескольких 201
- значения по умолчанию 206
- и методы 353, 355
- любое количество любого типа 430
- обработка интерпретатором 184
- передача по значению 220, 221
- передача по ссылке 220, 221
- позиционные и именованные 207
- список 426
- функций 190

Атрибуты (состояние)
- и классы 347
- инициализация 359
- и объекты 349, 351
- обзор 85
- поддержка сеансов в Flask 403
- поиск в словаре 404

Аутентификация 400

Б

Базы данных
- MariaDB 318
- MongoDB 594
- MySQL 318
  - достоинства 394
  - и DB-API 320
  - установка драйвера 321
  - установка драйвера MySQL-Connector/Python 322
  - установка сервера MySQL 319
- PostgreSQL 318
- SQLite 318

Библиотеки
- bokeh 591
- Kivy 596
- requests 545, 593
- threading 501
- tkinter 583

Булевы значения 152

В

Введение в Python DB-API 320

Веб-комик xkcd 603

Веб-приложения
- автоматический перезапуск 263
- вывод результатов 265

вычисление данных 266
декораторы функций 243, 245, 259
добавление надежности 491
запуск функций 244
и глобальные переменные 402
изменение 384
и обработка исключений 291, 454, 457, 458, 469, 473, 476
использование данных из запроса 263
коды состояния HTTP 258
обслуживание нескольких URL в одной функции 272
особенности работы 234
остановка 246, 256
отображение форм HTML 254
первый запуск 240
перезапуск 246, 256
перенаправление для устранения нежелательных ошибок 271
подготовка к развертыванию в облаке 274
последние штрихи 270
просмотр журналов 290
развертывание в службе PythonAnywhere 21, 565
размещение функциональности в Веб 245
создание объекта веб-приложения 242
тестирование 246, 256
технологии веб-разработки 592
установка и использование Flask 238
формы HTML 249, 262
цикл редактирование/остановка/запуск/проверка 260
что мы хотим от них 236
что происходит на веб-сервере 237

Веб-приложения с поддержкой баз данных
введение в Python DB-API 320
и обработка исключений 454, 457, 458, 469, 473, 476, 484
организация кода для работы с базой данных 337
программирование операций с базой данных и таблицами 332
создание базы данных и даблиц 323
установка драйвера MySQL-Connector/Python 322
установка драйвера базы данных 321
установка сервера MySQL 319
хранение данных 336

Веб-серверы 401
и процесс веб-приложения 402
коды состояния HTTP 258
обслуживание веб-приложений 234, 237

Великодушный пожизненный диктатор (BDFL) 600

ВерблюжийРегистр 348

| (вертикальная черта) 298

Вертикальная черта 298

Виртуальные окружения
технология venv 577

Вложенные словари 172

Вложенные функции 424, 436

Внутренние функции 424, 436

Внутренний адрес 127.0.0.1 247

Возвращаемые значения (функций) 192
и область видимости переменных 358
обработка интерпретатором 184

Время выполнения
обработка исключений 510

Встроенные функции
dir 295, 360
id 364
print
вывод объектов 365
необязательные аргументы 299
поведение по умолчанию 283
sorted 580

type 364
Вход и выход 411, 420
Вывод на экран
данных в читаемом формате 301
и обработка исключений 291
Выполнение кода
вызов функций 186
клавиша F5 187
параллельное 582
Выражения-генераторы 544, 546
встраивание в функции 547

Г

Гвидо ван Россум (Guido van Rossum) 600
Генераторы 529, 553, 540
выражения 544, 546
встраивание в функции 547
и кортежи 543
исследование 528
множеств 529, 542, 544
преобразование данных 520
преобразование шаблонного кода 527
пример компании Bahamas Buzzers 514
словарей 529, 535, 542, 544
списков 529, 542, 544
чтение данных CSV в виде списков 515
чтение данных CSV в словари 516
шаблон в коде 525
Глобальные переменные 402

Д

Даг Хэллман (Doug Hellmann) 581
Данные CSV
пример компании Bahamas Buzzers 514
чтение в виде списков 515
чтение в словари 516
Двоеточие (:)
и генераторы 542
и словари 134, 542
и списки 112
и функции 185, 198
Двоичный режим 283
Двойные подчеркивания, имена 242
Декораторы
@classmethod 578
@property 578
@staticmethod 578
Декораторы функций 421
route
добавление 253
изменение поведения 243
необязательные аргументы 259
описание 245
аргументы 259, 426, 428
добавление 253
и URL 243, 245
и адреса URL 432, 444
и диспетчеры контекста 443
изменение поведения 243
необходимые знания 421
создание 431
-= (декремента оператор) 142
Декремента оператор 142
Дзен Python 603
Динамические списки 86, 98
Динамическое присваивание переменным 84
Динамичные словари 150
Директивы
__slots__ 578

Директивы Jinja2
extends 250
Диспетчеры пакетов
Homebrew 319, 561
MacPorts 561
rpm 563
yum 563
Документирование функций 198
Драйверы баз данных
pymongo 594
Дэвид Бизли (David Beazley) 582

Ж

Журналы и журналирование
в одну строку с разделителями 298
и встроенная функция dir 295
изменение веб-приложения 386
исследование исходных данных 292
определение структуры 324
открыть, обработать и закрыть, методика обработки 286, 289
просмотр в веб-приложениях 290, 294

З

Запуск кода
Alt-P комбинация 154
Ctrl-P комбинация 154
, (запятая) 90, 159, 170
Запятая 90, 159, 170
Звездочки 426, 428
Значения
False 192
__name__ 242
True 192
Значения по умолчанию для аргументов 206

И

Имена с двойными подчеркиваниями 242
Имена с двумя подчеркиваниями 273
Именованные аргументы 207
Индексы и списки 99, 111
+= (инкремента оператор) 142
Инкремента оператор 142
Инсталлятор пакетов для Python (pip)
установка Flask 238
установка пакетов 218
установка плагина pep8 225
установка фреймворка тестирования pytest 225
Инструкции
def 183
значения по умолчанию для аргументов 206
и ключевое слово async 582
позиционные и именованные аргументы 207
else 153
if 153
import
Дзен Python 603
и модуль threading 501
и фреймворк Flask 241
местоположение 339
поиск модулей интерпретатором 210
совместное использование модулей 209
INSERT (SQL) 325, 499
return 192
возврат единственного значения 194
возврат нескольких значений 195
и круглые скобки 194
try...except 460, 467, 470, 477
with 346
и классы 341, 373
и метод split 518

и обработка исключений 479
и протокол управления контекстом 375
обработка исключений 487, 488
открыть, обработать и закрыть, методика обработки 283
просмотр журналов в веб-приложениях 290
повторное использование 338

Инструменты
pandas 591
pip (Package Installer for Python)
загрузка библиотеки requests 545
установка модуля pyreadline 560
PyLint 595
py.test 584
scikit-learn 591
для программирования 595
PyLint 595

Инструменты разработчика для тестирования 225

Инструменты тестирования для разработчика 584

Интегрированные среды разработки
Eclipse 589
Jupiter Notebook 590
PyCharm 589
WingWare 589

Интерактивная оболочка Python
ipython 588
ptpython 589
альтернативы 588
запуск из командной строки 211
копирование кода в редактор 93

Интерактивная оболочка Python (IDLE) 239, 589

Интерпретатор (Python)
CPython, эталонная реализация 597
IronPython 597
Jython 597
MicroPython 597
PyPy 597
Python 2 576
Python 3 576
рекомендации по использованию 576
альтернативные реализации 597
внутренний порядок 88
внутренний порядок следования записей 144
запуск из командной строки 211
и ключи в словарях 144
и функции 184
поиск модулей 210
синтаксические ошибки 93
чувствительность к регистру 152

Информационные бюллетени (Python) 602

Инъекция кода SQL 455, 458

Исключения
AttributeError 519
FileNotFoundError 459, 467
ImportError 212
InterfaceError 459, 477, 479
KeyError 151
NameError 357
PermissionError 462, 467
RuntimeError 459
SQLError 489
TypeError 355, 362, 366

K

Кавычки
и строки 113
строки 187

Каталоги хранилища сторонних пакетов 210, 213

каталог пакетов для Python (PyPI) 219

Каталог пакетов для Python (PyPI) 238, 593

Квадратные скобки []
  и генераторы 542
  и кортежи 169
  и словари 135, 177
  и списки 102, 110, 121
  списки 90
Клавиши
  F5 187
Классы 347
  ChainMap 581
  Counter 581
  Exception 463
  Flask 241
  object 360
  OrderedDict 581
  Thread 501
  и атрибуты 347
  и инструкция with 346, 373
  и методы 347, 354
  и объекты 348, 349, 351
  определение функциональности 349
  пустые 348, 480
  соглашения об именовании 348
  создание 346
Ключевые слова
  async 582
  await 582
  class 348
  pass 348
  return 183
Ключ/значение, пары
  обработка интерпретатором 144
Ключ/значение, пары (словари)
  введение 132
  добавление новых 137
  обзор 88
  создание во время выполнения 151
Кодекс поведения 601
Коды состояния (HTTP) 258
Коды состояния HTTP 271
Командная строка, запуск Python из 211, 226
Командные оболочки
  ipython 588
Команды
  cd 211
  describe log 325, 329
  help 102
  py 211, 226
  quit 211
  sudo 226, 238, 563
Комбинации клавиш
  Alt-P (Linux/Windows) 154
  Ctrl-C 256
  Ctrl-P (Mac) 154
Комментарии 183
Конец диапазона 112, 114
Конкатенации оператор) 579
Консоль MySQL 323
Конструкторы, методы 359
Конференции
  PyCon (Python Conference) 600
  PyCon, кодекс поведения 601
Кортежи
  введение 168
  и генераторы 543
  и списки 87, 168
  обзор 87
  определение в коде 168
  пустые 197
  с одним объектом 170
Круглые скобки ()
  аргументы функций 185
  и генераторы 542
  кортежи 168

Круглые скобки()
и инструкция return 194
^ (крышка) 228
крышка (^) 228

M

Межсайтовый скриптинг (XSS) 455, 458
Методы
append 94, 108, 306
close 281
copy 109
cursor 327
difference 163
__enter__ 374, 479
__exit__ 374, 479, 488
extend 100
format 579
__init__ 359, 366, 374, 479
insert 101
intersection 164, 195, 203
items 146
join 103, 294, 304
pop 99
remove 98
setdefault 155
split 304, 306, 515, 518
strip 518
union 163
курсора 499
Методы (HTTP)
GET 258
POST 258
Методы (поведение)
вызов 352
запуск веб-приложения 244
и аргументы 353, 355, 358
и атрибуты 358
изменение с помощью декораторов 243
и классы 347, 354
и объекты 349, 351, 358
и функции 352, 358
обзор 85
составление цепочек из вызовов методов 519
Механизм шаблонов Jinja2 250, 265
вычисление данных 266
Механизм шаблонов Jinjia2
вывод в читаемом формате 310, 312
Механизмы шаблонов 249
встраивание логики отображения в шаблон 311
отображение из Flask 253
подготовка кода к запуску 255
связь с веб-страницами 252
Множества
введение 159
и повторяющиеся объекты 89, 95, 159
обзор 89
общность 161, 164
объединение 161
определение в коде 159
пересечение 161
пустые 196
разность 161, 163
создание из последовательностей 160
эффективное создание 160
Модули
asyncio 582
collections 581
concurrent.futures 582
contextlib 373
csv 515
datetime 522
DB-API 14, 317
doctest 584

flask
класс Flask 241
словарь session 403
функция escape 293, 306
functools 438, 581
ImportError, исключение 212
itertools 581
matplotlib/seaborn 591
multiprocessing 582
pprint 175
pyreadline 560
random 210
requests 593
setuptools 214
sqlalchemy 594
sys 465
threading 501, 505, 582
turtle 583
unittest 584
virtualenv 577
добавление в каталоги хранилища сторонних пакетов 214
импортирование 209
и функции 209
совместное использование 219
создание 209

Н

Наоми Чедер (Naomi Ceder) 601
Наука о данных 591
Начало диапазона 112, 114
Неизменяемые списки (кортежи) 87
Неупорядоченные структуры данных 88
Номер порта протокола 240, 247
* нотация 426
** нотация 428

О

Область видимости переменных 357
Обработка исключений
времени выполнения 459, 510. См. конкретные исключения; См. конкретные исключения
встроенные исключения 463
вывод на экран 291
и базы данных 454, 457, 458, 469, 473, 476, 484
и веб-приложения 291, 454, 457, 458, 469, 473, 476
и диспетчеры контекста 476
и инструкция with 479, 487, 488
и функции 457, 458
обработчик всех исключений 464, 467
синтаксические ошибки. См. конкретные исключения
создание собственных исключений 480
сообщения об ошибках PEP 8. См. конкретные исключения
\\ (обратный слеш) 113
Обратный слеш 113
Объект запроса (Flask) 295, 360
Объектно-ориентированное программирование (ООП) 347, 360, 578
Объекты
Markup (Flask) 293
request (Flask) 262
веб-приложений 242
добавление в списки 100
и атрибуты 349, 351, 358
извлечение из списков 99
и классы 348, 349, 351
и методы 349, 351, 358
и пары ключ/значение 132
обзор 84

определение представления 364
повторяющиеся 89, 95
последовательности объектов 160
создание 348, 359
удаление из списков 98
функции 422, 431, 433

+ (оператор конкатенации) 579

= (оператор присваивания) 91, 108

+= (оператор увеличения) 354

* (оператор умножения) 123

Операторы
in
и множества 161
и словари 151
in и списки 92, 95
not in 95, 154
декремента (-=) 142
инкремента (+=) 142
конкатенации) 579
присваивания 91, 108
проверка вхождения 153
проверки вхождения 92, 95
тернарный 153
увеличения 354
умножения 123

Основные разработчики Python 600

Открыть, обработать и закрыть, методика обработки 281
вызов функции журналирования 286, 289
и инструкция with 283
чтение данных из файлов 282

отладка 260

Отладка 585
pdb, отладчик 585

Отступы
в функциях 183

П

Пакеты
numpy 591
scipy 591

Параллельное выполнение, варианты 501

Параллельное выполнение кода 582

Передача аргументов по значению 220, 221

Передача аргументов по ссылке 220, 221

Переменные
глобальные 402
динамическое присваивание 84
инициализация 359
и функции 357
область видимости 357
присваивание значений 84
присваивание объектов 84

Плагины
pep8 225

Повторяющиеся объекты и множества 89, 95

Подкасты (Python) 602

Подсказки типов (аннотации) 198

Подсчет частоты
введение 138
выбор структуры данных 140
изменение счетчика 141
инициализация 141

Позиционные аргументы 207

Последовательности объектов 160

Предложения по развитию Python (PEP) 189
описание DB-API 320
проверка на соответствие 224, 584
рекомендуемая длина строки 299

Преобразование
данных 520

Присваивания, оператор 91, 108

Пробельные символы 228, 518, 554

Протокол HTTP
  коды состояния 258

Протокол передачи гипертекста (HyperText Transfer Protocol, HTTP) 401

Протокол управления контекстом 341, 346, 374
  выполнение заключительных операций 374, 381
  выполнение настройки 374, 379
  и декораторы функций 443
  инициализация диспетчера контекста 374
  и обработка исключений 476
  повторное обсуждение веб-приложения 384
  создание диспетчера контекста 373, 375
  создание класса диспетчера контекста 376
  тестирование диспетчера контекста 382

/ (прямой слеш) 243

Прямой слеш 243

Пустая операция 348

Пустые классы 348, 480

Пустые кортежи 197

Пустые множества 196

Пустые словари 140, 172, 197

Пустые списки 91, 94, 197

Пустые строки 193

P

Разделители 113, 298

Распространение пакетов 214

Редактирования окно 93, 186, 187

Результаты, возврат из функций 192

Руководства
  Python Packaging Authority 219

C

Сеансы 403
  и состояние 403
  управление входом и выходом 411

Семантика вызова, передача аргументов по значению 220, 221

Семантика вызова, передача аргументов по ссылке 220, 221

Сетевое имя localhost 247

@ символ 243

символ 183

Символы
  стрелка 198

% синтаксис 250

Синтаксис срезов
  и списки 121
  и цикл for 123

Синтаксис точечной нотации 94

% синтаксис форматирования 579

Синтаксические ошибки 93

Словари
  form (Flask) 262
  session (модуль flask) 403
  аргументов 428
  введение 139
  внутри словарей 172
  динамически изменяют размеры 137
  динамичная структура данных 150
  изменение порядка вывода 145
  и квадратные скобки 135
  и пары ключ/значение 151

и подсчет частоты 138
итерации по записям 143, 146
итерации по ключам и значениям 144
обзор 88
определение в коде 134
пары ключ/значение 88, 132
проверка вхождения 153
простота чтения 132
пустые 140, 172, 197
списков 521
чтение данных CSV 516

Словарь ImmutableMultiDict 297

Служба PythonAnywhere 21, 565
выгрузка файлов в облако 274, 568
извлечение и установка кода 569
настройка веб-приложения 572
опробование развернутого веб-приложение 573
подготовка веб-приложений 566
регистрация 567
создание начального веб-приложения 570

Создание объекта 359

Создание экземпляров 348

Сообщения
информационные 258
коды состояния HTTP 258, 271
об ошибках на стороне клиента 258
об ошибках на стороне сервера 258
об успехе 258
перенаправления 258

Сортировка 580

Сохранение данных
в текстовых файлах 281

Списки
аргументов 426
динамические 86, 98
добавление объектов 100
заполнение во время выполнения 94
извлечение объектов 99
и кортежи 87, 168
когда не следует использовать 126
копирование существующих 108
литеральные 90
начало и конец 114
обзор 86, 90, 125
обход 115
оператор присваивания 91, 108
проверка вхождения в 92, 95
пустые 91, 94, 197
работа со списками 92, 107
синтаксис срезов 113
словарей 521
создание литеральных 91
сортировка 162
удаление объектов из 98
форма записи с квадратными скобками 90, 102, 110, 121
форма записи срезов 121
чтение данных CSV 515
шаблон в коде 526

Срезов синтаксис
и списки 113

Стандартная библиотека 182
варианты параллельного выполнения 501
дополнительные сведения 581, 583
недопустимость удаления/добавления модулей 214

Строка документации
добавление 204
добавление информации 198
обновление 201

Строки
в кавычках 187
и кавычки 113
и пары ключ/значение 132

и пробельные символы 518
объединение 103, 294, 304
превращение в списки символов 114
присваивание переменным 84
пустые 193
разделение 304
форматирование 579

Строки документации 183
добавление 187

Структуры данных
встроенные 86, 197
копирование 109
сложные 171, 302

Т

Таблицы 332

Теги
<form> 258
<table> 310
<td> 310
<th> 310
<tr> 310

Текстовые редакторы
vim 589

Текстовые файлы, сохранение данных 281

Текстовый режим 283

Текущий рабочий каталог 210

Тернарный оператор 153

Тестирование
автоматизированное 584
инструменты для разработчика 584

Технологии веб-разработки 592

Технология виртуальных окружений venv 577

Тим Петерс (Tim Peters) 603

Точечная нотация 352

У

<> (угловые скобки) 292

Угловые скобки 292

Умножения, оператор 123

Упорядоченные структуры данных 86

Управление памятью 98

Установка Python 3
в Linux 563
в Mac OS X 561
в Windows 558

Утилиты
apt-get 563

Ф

Файлы
README.txt 215
ZIP 216

Фигурные скобки {}
и генераторы 542
и множества 159
и словари 140
механизмы шаблонов 250

Фонд содействия развитию Python (PSF) 600

Форматирование
данных 520
строк 579

Формы HTML
вывод результатов 265
доступ из Flask 262
конструирование 249
отображение 254
отображение шаблонов из Flask 253
перенаправление для устранения нежелательных ошибок 270

тестирование 255
Фреймворк Django 239
Фреймворк Flask 592
доступ к данным HTML-форм 262
запуск веб-приложения 240
и механизм сеансов 403
и многопоточное выполнение 507
механизм шаблонов Jinja2 250, 265
механизм шаблонов Jinjia2 310, 312
обслуживание нескольких URL в одной функции 272
объект Markup 293
объект request 262
объект запроса 295, 360
описание 239
отображение шаблонов 253
режим отладки 260
создание объекта веб-приложения 242
тестирование веб-приложений 256
установка 238
Фреймворки
Django 592
Flask 592
Фреймворк тестирования pytest 225
Функции 183
app.run() 243, 247, 253
bool, встроенная функция 192
combinations (модуль itertools) 581
connect 327
escape (модуль flask) 293, 306
list, встроенная функция 162
open 281
partial (модуль functools) 581
permutations (модуль itertools) 581
pprint (модуль pprint) 175
print встроенная функция
доступ к значениям в словарях 144
product (модуль itertools) 581
randint (модуль random) 210
redirect (Flask) 271
render_template (Flask) 253, 270
setyp (модуль setuptools) 215
set встроенная функция 160
set, встроенная функция 196, 203
sorted встроенная функция
и множества 159
и словари 145
sorted, встроенная функция
и множества 162
type, встроенная функция 168
аргументы 190
вложенные 424, 436
возврат из функций 425
возврат результата 192
вызов 186
вызов переданной функции 423
добавление нескольких аргументов 201
документирование 198
именование 185, 201
и методы 352
и модули 209
и обработка исключений 457
и переменные 357
обслуживание нескольких URL 272
передача в функции 422
повторное использование кода 182, 209
редактирование 186
рекомендации 189
соглашения об именовании 348
создание 185, 202
строки в кавычках 187
Функции, встроенные
input 96
len 94
Функции-объекты 422, 431, 433
Функция
bool 192

Ц

Цикл for 540
  и списки 515. См. генераторы
  и срезы. См. генераторы
  шаблон в коде 525

Ч

Числа
  присваивание переменным 84
Чтение
  данных CSV в виде списков 515
  данных CSV в словари 516
  данных из файлов 282
Чувствительность к регистру и соглашения 152, 348

Ш

Шаг диапазона 112, 115

Э

Экранирование символов 113
Экранированные последовательности 293

M

MySQL
  и обработка исключений 454, 457, 458, 469, 473, 476, 484

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Пол Бэрри

ИЗУЧАЕМ ПРОГРАММИРОВАНИЕ НА PYTHON

Директор редакции *Е. Капьёв*
Ответственный редактор *Е. Истомина*
Художественный редактор *А. Шуклин*

ООО «Издательство «Э»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

Өндіруші: «Э» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86.
Тауар белгісі: «Э»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251 58 12 вн. 107.
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта Өндіруші «Э»

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Э»

Өндірген мемлекет: Ресей
Сертификация қарастырылмаған

Подписано в печать 29.06.2017. Формат 84x108$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 65,52.
Тираж экз. Заказ

Знак информационной продукции согласно Федеральному закону от 29.12.2010 г. № 436-ФЗ
12+

ISBN 978-5-699-98595-1

В электронном виде книги издательства вы можете купить на www.litres.ru

ЛитРес: один клик до книг

Оптовая торговля книгами Издательства «Э»:
142700, Московская обл., Ленинский р-н, г. Видное,
Белокаменное ш., д. 1, многоканальный тел.: 411-50-74.

По вопросам приобретения книг Издательства «Э» зарубежными оптовыми покупателями обращаться в отдел зарубежных продаж
International Sales: International wholesale customers should contact Foreign Sales Department for their orders.

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении, *обращаться по тел.: +7 (495) 411-68-59, доб. 2261.*

Оптовая торговля бумажно-беловыми и канцелярскими товарами для школы и офиса:
142702, Московская обл., Ленинский р-н, г. Видное-2,
Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).

Полный ассортимент книг издательства для оптовых покупателей:
Москва. Адрес: 142701, Московская область, Ленинский р-н,
г. Видное, Белокаменное шоссе, д. 1. Телефон: +7 (495) 411-50-74.
Нижний Новгород. Филиал в Нижнем Новгороде. Адрес: 603094,
г. Нижний Новгород, улица Карпинского, дом 29, бизнес-парк «Грин Плаза».
Телефон: +7 (831) 216-15-91 (92, 93, 94).
Санкт-Петербург. ООО «СЗКО». Адрес: 192029, г. Санкт-Петербург, пр. Обуховской Обороны,
д. 84, лит. «Е». Телефон: +7 (812) 365-46-03 / 04. **E-mail:** server@szko.ru
Екатеринбург. Филиал в г. Екатеринбурге. Адрес: 620024,
г. Екатеринбург, ул. Новинская, д. 2щ. Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08).
Самара. Филиал в г. Самаре. Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е».
Телефон: +7 (846) 269-66-70 (71…73). **E-mail:** RDC-samara@mail.ru
Ростов-на-Дону. Филиал в г. Ростове-на-Дону. Адрес: 344023,
г. Ростов-на-Дону, ул. Страны Советов, 44 А. Телефон: +7(863) 303-62-10.
Центр оптово-розничных продаж Cash&Carry в г. Ростове-на-Дону. Адрес: 344023,
г. Ростов-на-Дону, ул. Страны Советов, д.44 В. Телефон: (863) 303-62-10. Режим работы: с 9-00 до 19-00.
Новосибирск. Филиал в г. Новосибирске. Адрес: 630015,
г. Новосибирск, Комбинатский пер., д. 3. Телефон: +7(383) 289-91-42.
Хабаровск. Филиал РДЦ Новосибирск в Хабаровске. Адрес: 680000, г. Хабаровск,
пер.Дзержинского, д.24, литера Б, офис 1. Телефон: +7(4212) 910-120.
Тюмень. Филиал в г. Тюмени. Центр оптово-розничных продаж Cash&Carry в г. Тюмени.
Адрес: 625022, г. Тюмень, ул. Алебашевская, 9А (ТЦ Перестройка+).
Телефон: +7 (3452) 21-53-96/ 97/ 98.
Краснодар. Обособленное подразделение в г. Краснодаре
Центр оптово-розничных продаж Cash&Carry в г. Краснодаре
Адрес: 350018, г. Краснодар, ул. Сормовская, д. 7, лит. «Г». Телефон: (861) 234-43-01(02).
Республика Беларусь. Центр оптово-розничных продаж Cash&Carry в г.Минске. Адрес: 220014,
Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto».
Телефон: +375 17 251-40-23; +375 44 581-81-92. Режим работы: с 10-00 до 22-00.
Казахстан. РДЦ Алматы. Адрес: 050039, г. Алматы, ул.Домбровского, 3 «А».
Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99).
Украина. ООО «Форс Украина». Адрес: 04073, г.Киев, Московский пр-т, д.9.
Телефон: +38 (044) 290-99-44. **E-mail:** sales@forsukraine.com

**Полный ассортимент продукции Издательства «Э»
можно приобрести в магазинах «Новый книжный» и «Читай-город».**
Телефон единой справочной: 8 (800) 444-8-444. Звонок по России бесплатный.

В Санкт-Петербурге: в магазине «Парк Культуры и Чтения БУКВОЕД», Невский пр-т, д.46.
Тел.: +7(812)601-0-601, www.bookvoed.ru

Розничная продажа книг с доставкой по всему миру. Тел.: +7 (495) 745-89-14.

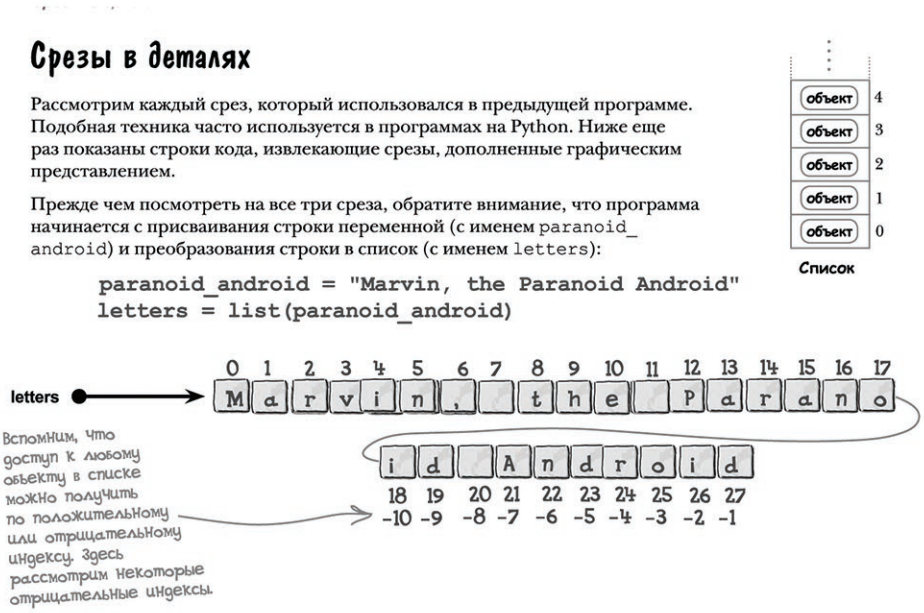
BOOK24.RU
ИНТЕРНЕТ-МАГАЗИН
ИНТЕРНЕТ-МАГАЗИН
BOOK24.RU

ИЗУЧАЕМ ПРОГРАММИРОВАНИЕ НА PYTHON

Программирование/Python

О чем эта книга?

Надоело продираться через дебри малопонятных самоучителей по программированию? С этой книгой вы без труда усвоите азы Python и научитесь работать со структурами и функциями. В ходе обучения вы создадите свое собственное веб-приложение и узнаете, как управлять базами данных, обрабатывать исключения, пользоваться контекстными менеджерами, декораторами и генераторами. Все это и многое другое – во втором издании «Изучаем программирование на Python».



Почему эта книга столь не похожа на другие?

Основанная на новейших исследованиях работы мозга, эта книга имеет максимально наглядный формат, благодаря которому вы усвоите больше информации, чем при чтении скучных колонок текста. Так зачем же тратить время и силы, сражаясь с непонятными описаниями, если можно заставить свой мозг работать эффективнее!

«Эта книга поможет превратить обычно серьезный процесс изучения программирования в ненавязчивую игру. Вы сами не заметите, как, доверившись оригинальному подходу авторов, начнете программировать на одном из самых популярных языков».

– Кирилл Жвалов, Сооснователь Moscow Coding School

«Книга по Python должна быть такой же веселой, как и сам язык. В «Изучаем программирование на Python» Пол Бэрри приглашает читателей в насыщенное и увлекательное путешествие по этому языку, в конце которого они будут готовы программировать на Python».

– Эрик Фриман, доктор компьютерных наук, бывш. технический директор Disney Online

O'REILLY®

ISBN 978-5-699-98595-1