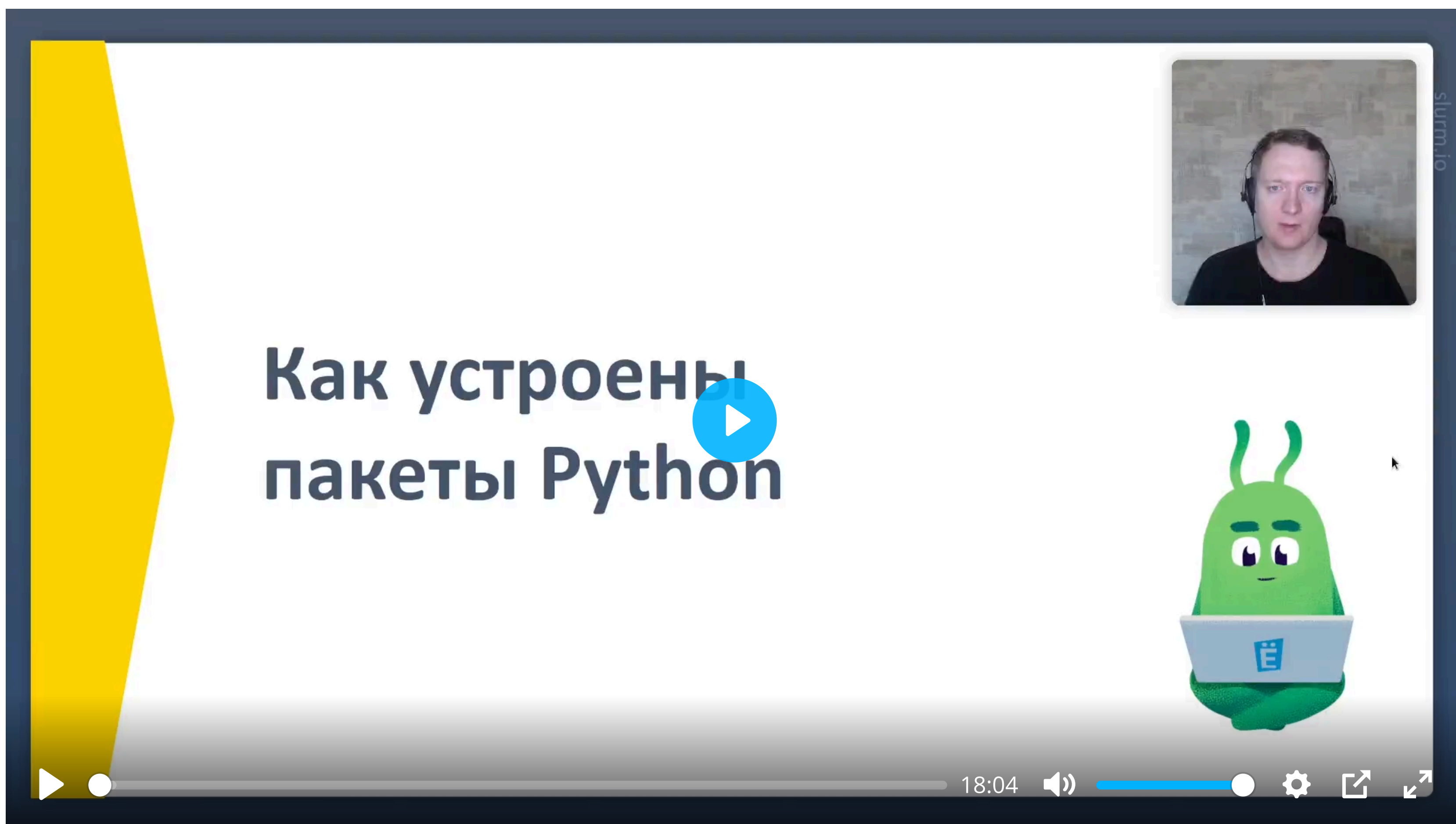
2.2 Инструменты экосистемы Python для работы с данными



Текстовая расшифровка видео:

КАК УСТРОЕНЫ ПАКЕТЫ PYTHON

План:

- Пакеты и исходники;
- Структурируем схему пакета;
- Setup.py;
- MANIFEST.in;
- Собираем бинарный дистрибутив;
- Manylinux.

Пакеты и исходники

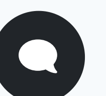
Ранее говоря о mkvirtualenv, мы запускали команду (~$ pip install -U pip wheel setuptools) с установкой метапакетов, инсталляторов. Инсталляторы переходят в глобальные репозитории пакетов, которые находятся на сайте pypi.org, скачивают соответствующий пакет в виде архива и распаковывают файловую систему.

Пакеты бывают (в общем случае) двух типов:

- Архивы с кодом (чаще всего формата .tar.gz);
- Бинарные дистрибутивы (формата whl).

Нюанс: пакеты могут содержать куски кода на других языках, в том числе на компилируемых (C, C++). Тогда при установке из архивов код должен скомпилироваться, результат и успешность компиляции будет зависеть от того, что стоит в системе.

При сборке современных пакетов на Python, которые имеют внешние куски кода на других языках, рекомендуется использовать второй тип пакетов – Бинарные дистирибутивы.



Главный файл пакета – **setup.py** (или pyproject.toml).

Интерпретатор по умолчанию ищет пакеты в **/usr/lib/python3.11**, либо же в **venv/lib/python3.11/site-packages**. Когда мы запускаем интерпретатор, он имеет определенную область видимости и пакеты будет искать именно по этим путям.

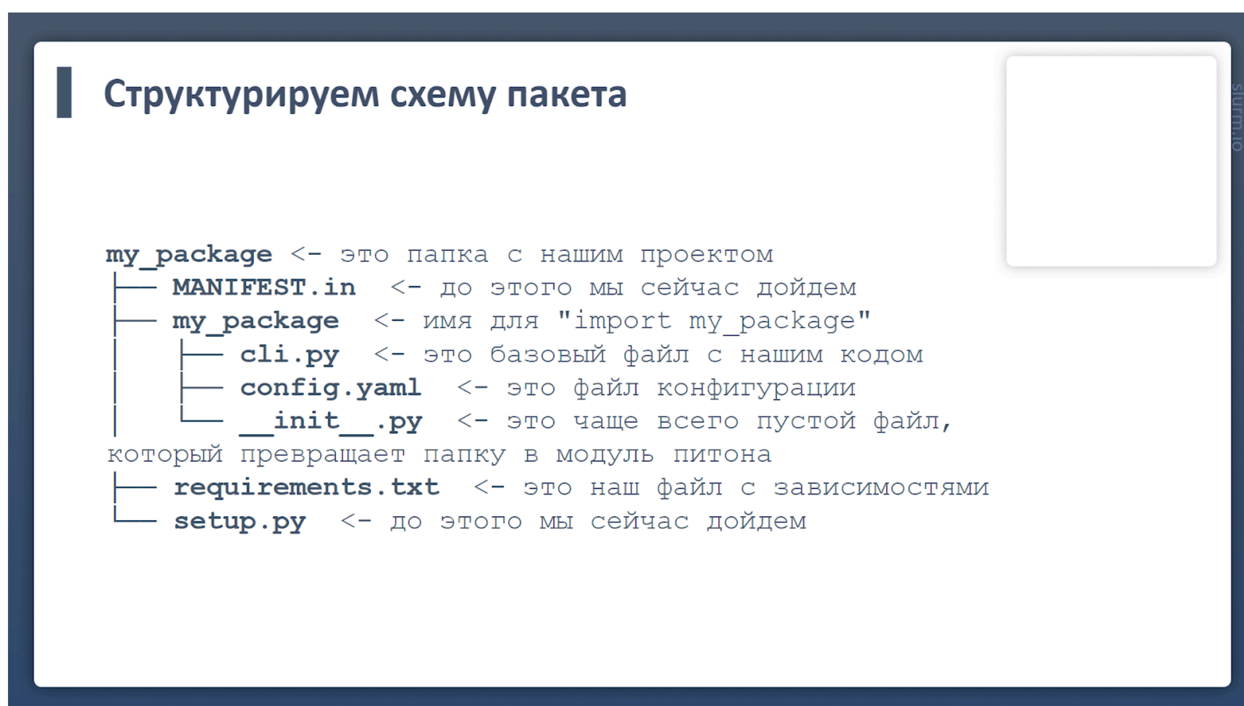
Можно передавать дополнительные пути для поиска с помощью переменной окружения PYTHONPATH или добавляя в sys.path.

Одна из важных проблем в Python:

Когда мы ставим зависимости через pip install, единственный способ понять, какие есть зависимости у пакета – скачать его и распаковать. Заранее не получится высчитать граф зависимостей для пакета, который мы ставим.

Структурируем схему пакета

Для создания собственного Python-пакета необходимо создать подобную структуру файлов и каталогов:



Здесь есть папка с проектом, в ней находятся несколько дополнительных метафайлов: файл с зависимостями (requirements.txt), файл MANIFEST.in.

Сам код пакета лежит внутри еще одного каталога, где имя каталога по умолчанию будет использоваться в команде «Импорт в коде».

В случае с Python, чтобы каталог имел внутри себя импортируемый код, в нем должен находиться файл **__init__.py**

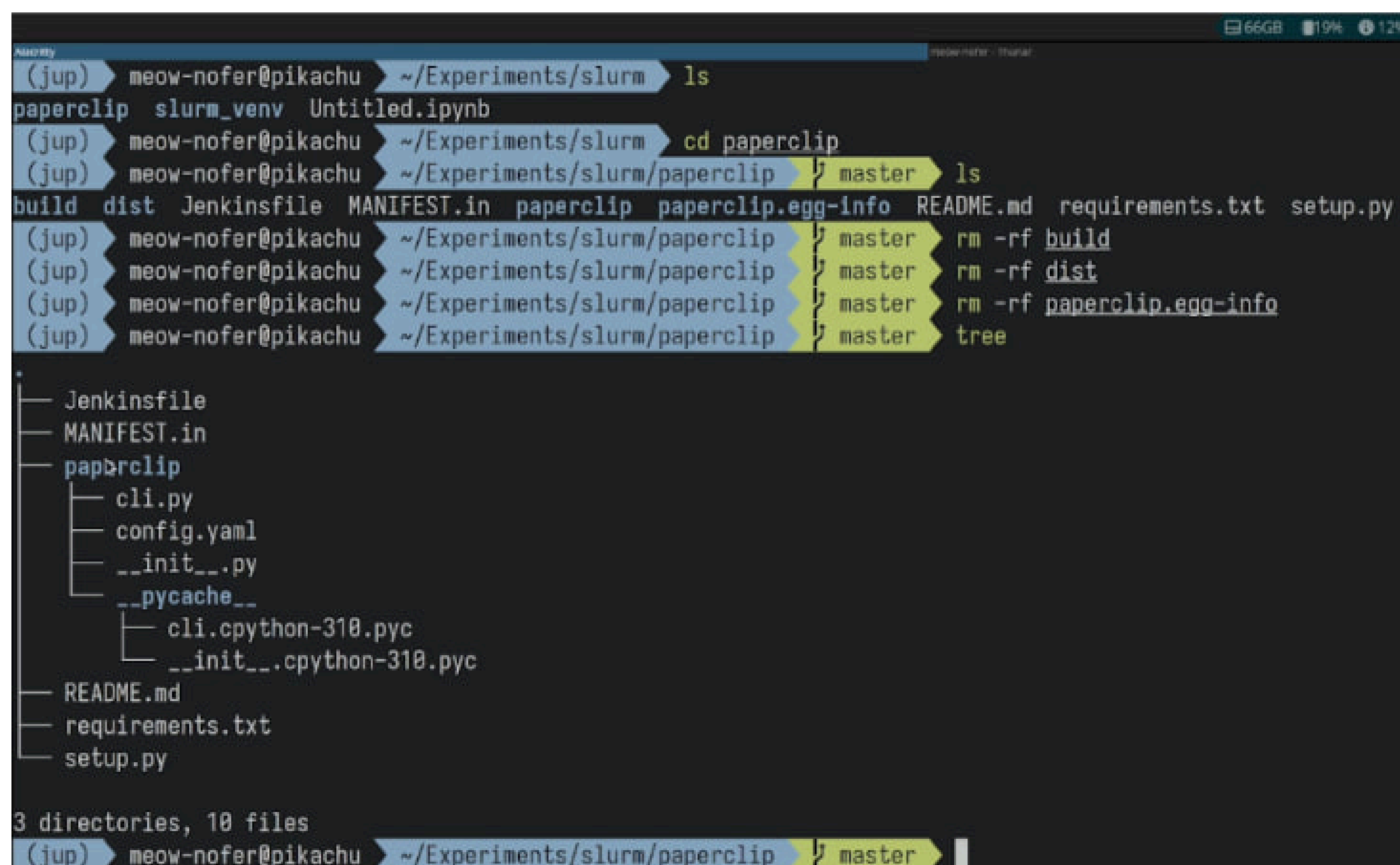
Setup.py

```
1. import os
2. import os.path
3.
4. from setuptools import find_packages
5. from setuptools import setup
6.
7. def find_requires():
8.   dir_path = os.path.dirname(os.path.realpath(__file__))
9.   requirements = []
10.   with open('{0}/requirements.txt'.format(dir_path), 'r') as reqs:
11.       requirements = reqs.readlines()
12.   return requirements
13.
14. if __name__ == "__main__":
15.   setup(
16.       name="my_package",
17.       version="0.0.1",
18.       description='my cool package',
19.       packages=find_packages(),
20.       install_requires=find_requires(),
21.       include_package_data=True,
22.       entry_points={
23.           'console_scripts': [
24.               'my_command = my_package.cli:main',
25.           ],
26.       },
27.   )
```

Основная цель Setup.py – вызвать конкретную функцию setup из модуля setuptools. Туда мы прописываем ряд информации о том, что есть в нашем пакете (версия, название, описание). Далее мы помещаем все Python-файлы в пакет (работает через «my_package»), после чего указываем зависимости пакета.

Когда мы ставим пакет в систему, хочется, чтобы в пространстве видимости появлялась команда, при запуске которой запускался бы код из пакета. Данная ситуация решается через **entry_points и console_scripts**. Так, при установке виртуального пакета появится команда «My_command», которая при запуске в консоли из модуля «my_package» и под модулем «cli», вызовет функцию «main».

В качестве примера может служить проект «Paperclip», который соответствует указанной структуре:



Setup.py выглядит следующим образом:

```
setup.py
1 import os
2 import os.path
3
4 from setuptools import find_packages
5 from setuptools import setup
6
7
8 def find_requires():
9     dir_path = os.path.dirname(os.path.realpath(__file__))
10    requirements = []
11    with open('{0}/requirements.txt'.format(dir_path), 'r') as reqs:
12        requirements = reqs.readlines()
13    return requirements
14
15
16 if __name__ == "__main__":
17    setup(
18        name="paperclip",
19        version="0.0.1",
20        description='my cool package',
21        packages=find_packages(),
22        install_requires=find_requires(),
23        include_package_data=True,
24        entry_points={
25            'console_scripts': [
NORMAL  setup.py                                        python
"setup.py" 30L, 705B
[0] 0:zsh*
```

MANIFEST.in

В проекте могут быть файлы, не относящиеся к Python (request, конфиги) по умолчанию. Если мы попробуем собрать пакет, не ссылаясь на эти файлы, то они не попадут в него. Setuptools добавляет по умолчанию только файлы Python. Для того, чтобы подключить эти файлы к пакету следует использовать следующую комбинацию:

```
21.        include_package_data=True,
```

В setuptools используется специальный флаг «include_package_data=True». Это означает, что присутствуют файлы, не относящиеся к Python. Эта директива сообщает setuptools о необходимости обратить внимание на файлы Manifest.In. В файле Manifest.In. при помощи простого синтаксиса прописываем, какие файлы хотим включить в проект.

Примеры:

- include *requirements.txt
- recursive-include my_package *

В данном проекте, если мы посмотрим на Manifest.In., включается файл requirements.txt и recursive-include из всего каталога paperclip, туда попадет еще и файл config.yaml при сборке пакетов:

```
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master ls
build  dist  Jenkinsfile  MANIFEST.in  paperclip  paperclip.egg-info  README.md  requirements.txt  setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf build
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf dist
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf paperclip.egg-info
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master tree
.
├── Jenkinsfile
├── MANIFEST.in
├── paperclip
│   ├── cli.py
│   ├── config.yaml
│   ├── __init__.py
│   └── __pycache__
│       ├── cli.cpython-310.pyc
│       └── __init__.cpython-310.pyc
├── README.md
├── requirements.txt
└── setup.py

3 directories, 10 files
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master vim setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master ls
Jenkinsfile  MANIFEST.in  paperclip  README.md  requirements.txt  setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master cat MANIFEST.in
include *requirements.txt
recursive-include paperclip *
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master
[0] 0:zsh*
```

Что нужно для сбора пакета:

- Setup.py;
- MANIFEST.in (если есть внешние файлы).
- Запускаем команду **~$ python setup.py sdist**

Запускаем **setup.py sdist:**

```
build  dist  Jenkinsfile  MANIFEST.in  paperclip  paperclip.egg-info  README.md  requirements.txt  setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf build
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf dist
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master rm -rf paperclip.egg-info
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master tree
.
├── Jenkinsfile
├── MANIFEST.in
├── paperclip
│   ├── cli.py
│   ├── config.yaml
│   ├── __init__.py
│   └── __pycache__
│       ├── cli.cpython-310.pyc
│       └── __init__.cpython-310.pyc
├── README.md
├── requirements.txt
└── setup.py

3 directories, 10 files
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master vim setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master ls
Jenkinsfile  MANIFEST.in  paperclip  README.md  requirements.txt  setup.py
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master cat MANIFEST.in
include *requirements.txt
recursive-include paperclip *
(jup) meow-nofer@pikachu ~/Experiments/slurm/paperclip  master python setup.py sdist

[0] 0:zsh*
```

Происходит следующее:

```
reading manifest template 'MANIFEST.in'
writing manifest file 'paperclip.egg-info/SOURCES.txt'
running check
creating paperclip-0.0.1
creating paperclip-0.0.1/paperclip
creating paperclip-0.0.1/paperclip.egg-info
creating paperclip-0.0.1/paperclip/__pycache__
copying files to paperclip-0.0.1...
copying MANIFEST.in -> paperclip-0.0.1
copying README.md -> paperclip-0.0.1
copying requirements.txt -> paperclip-0.0.1
copying setup.py -> paperclip-0.0.1
copying paperclip/__init__.py -> paperclip-0.0.1/paperclip
copying paperclip/cli.py -> paperclip-0.0.1/paperclip
copying paperclip/config.yaml -> paperclip-0.0.1/paperclip
copying paperclip.egg-info/PKG-INFO -> paperclip-0.0.1/paperclip.egg-info
copying paperclip.egg-info/SOURCES.txt -> paperclip-0.0.1/paperclip.egg-info
copying paperclip.egg-info/dependency_links.txt -> paperclip-0.0.1/paperclip.egg-info
copying paperclip.egg-info/entry_points.txt -> paperclip-0.0.1/paperclip.egg-info
copying paperclip.egg-info/requires.txt -> paperclip-0.0.1/paperclip.egg-info
copying paperclip.egg-info/top_level.txt -> paperclip-0.0.1/paperclip.egg-info
copying paperclip/__pycache__/__init__.cpython-310.pyc -> paperclip-0.0.1/paperclip/__pycache__
copying paperclip/__pycache__/cli.cpython-310.pyc -> paperclip-0.0.1/paperclip/__pycache__
Writing paperclip-0.0.1/setup.cfg
creating dist
Creating tar archive
removing 'paperclip-0.0.1' (and everything under it)
 (jup)  meow-nofer@pikachu  ~/Experiments/slurm/paperclip  master
[0] 0:zsh*
```

Появится каталог «egg-info» (промежуточная информация) и каталог «dist», в котором будет файл с исходниками – paperclip-0.0.1.tar.gz – тот самый пакет в виде архива с конкретной версией, который можно уже ставить с «pip install»:

```
Creating tar archive
removing 'paperclip-0.0.1' (and everything under it)
 (jup)  meow-nofer@pikachu  ~/Experiments/slurm/paperclip  master  tree
.
├── dist
│   └── paperclip-0.0.1.tar.gz
├── Jenkinsfile
├── MANIFEST.in
├── paperclip
│   ├── cli.py
│   ├── config.yaml
│   ├── __init__.py
│   └── __pycache__
│       ├── cli.cpython-310.pyc
│       └── __init__.cpython-310.pyc
├── paperclip.egg-info
│   ├── dependency_links.txt
│   ├── entry_points.txt
│   ├── PKG-INFO
│   ├── requires.txt
│   ├── SOURCES.txt
│   └── top_level.txt
├── README.md
├── requirements.txt
└── setup.py

5 directories, 17 files
 (jup)  meow-nofer@pikachu  ~/Experiments/slurm/paperclip  master
[0] 0:zsh*
```

Собираем бинарный дистрибутив

Для создания бинарного дистрибутива существует дополнительный метапакет «whell» и дополнительная команда «setup.py bdist_whell».

Так, бинарная сборка создает еще один каталог с метаинформацией (для хранения промежуточных артефактов сборки, а также создается еще один пакет, имеющий формат whl, конкретную версию и пр.

Поскольку в данном пакете есть только Python-код и нам не нужно собирать дополнительные зависимости, whl получится универсальным. Однако, чем больше платформоспецифичных вещей мы используем, тем больше шансов, что при попытке собрать whl, собранный бинарный дистрибутив будет платформозависимым. Если мы запустим эту команду «setup.py bdist_whell» на macOS, у нас может собраться whl, которая будет не «none», а «Darwin».

Manylinux

manylinux

Older archives: https://groups.google.com/forum/#!forum/manylinux-discuss

The goal of the manylinux project is to provide a convenient way to distribute binary Python extensions as wheels on Linux. This effort has produced PEP 513 (manylinux1), PEP 571 (manylinux2010), PEP 599 (manylinux2014) and PEP 600 (manylinux_x_y).

PEP 513 defined `manylinux1_x86_64` and `manylinux1_i686` platform tags and the wheels were built on Centos5. Centos5 reached End of Life (EOL) on March 31st, 2017.

PEP 571 defined `manylinux2010_x86_64` and `manylinux2010_i686` platform tags and the wheels were built on Centos6. Centos6 reached End of Life (EOL) on November 30th, 2020.

PEP 599 defines the following platform tags:

- `manylinux2014_x86_64`
- `manylinux2014_i686`
- `manylinux2014_aarch64`
- `manylinux2014_armv7l`
- `manylinux2014_ppc64`
- `manylinux2014_ppc64le`
- `manylinux2014_s390x`

Wheels are built on CentOS 7 which will reach End of Life (EOL) on June 30th, 2024.

PEP 600 has been designed to be "future-proof" and does not enforce specific symbols and a specific distro to build. It only states that a wheel tagged `manylinux_x_y` shall work on any distro based on `glibc>=x.y`. The manylinux project supports:

- `manylinux_2_24` images for `x86_64`, `i686`, `aarch64`, `ppc64le` and `s390x`.
- `manylinux_2_28` images for `x86_64`, `aarch64`, `ppc64le` and `s390x`.

Wheel packages compliant with those tags can be uploaded to PyPI (for instance with twine) and can be installed with pip:

Что, если мы хотим включить в пакет код на другом языке, на котором мы оптимизировали узкое место в расчетах? Для этого существует проект, который называется «Manylinux»

Manylinux – это виртуализированная среда, существующая как в виде виртуальных машин, так и в виде докер-контейнеров, в которых расположено минимальное окружение с набором библиотек, которые можно статически линковать.

Эти библиотеки гарантируют следующее: пакет, который вы соберете, линкуя с этими библиотеками, с наибольшей вероятностью заработает на большинстве современных дистрибутивов.

Слёрм ©

+7 (495) 248-05-80

Лицензия №ДЛ-1368 от 22.08.2019

Политика конфиденциальности

Публичная оферта