

a) I've created two solutions, each using a slightly different approach. Let's first talk about the one in `solution_oo.py`.

If you want to use composition instead of inheritance in this case, it means that we need to create a class that *contains* the `CloudProvider` object. I've modified the `ACCloud` class and turned it into a dataclass that contains both the `CloudProvider` object and the bucket name. `ACCloud` still contains the `find_files` method and accesses the `CloudProvider` object's `filter_by_query` function.

Second, I added a function that helps create `CloudProvider` objects. This way, the `ACCloud` class is not responsible for creating its own objects, which generally is a good thing (see the "Separate Creation From Use" lesson in this course). In order to fix the second layer of inheritance, having two constants to store the bucket name and region and then use that as a default value already does the job.

You can take this a step further by removing the `ACCloud` class altogether and simply replacing it by a function. You can find this version of the solution in `solution_fn.py`. The main difference is that the class is no longer needed, although the feature that overrides the `bucket_name` and `region` is now gone. It's not really needed anymore though since you already pass these as an argument to the function.

b) In order to remove the dependency of `find_files` on `CloudProvider`, we need to introduce abstraction. I did this in `solution_fn_abstract.py`. I defined a type to achieve this and passed a `filter_fn` function as an argument. `find_files` is now no longer directly dependent on `CloudProvider` and you can still easily call it if you have an `CloudProvider` object:

```
cloud_provider = create_cloud_provider()
find_files(cloud_provider.filter_by_query, bucket_name, query, max_result)
```