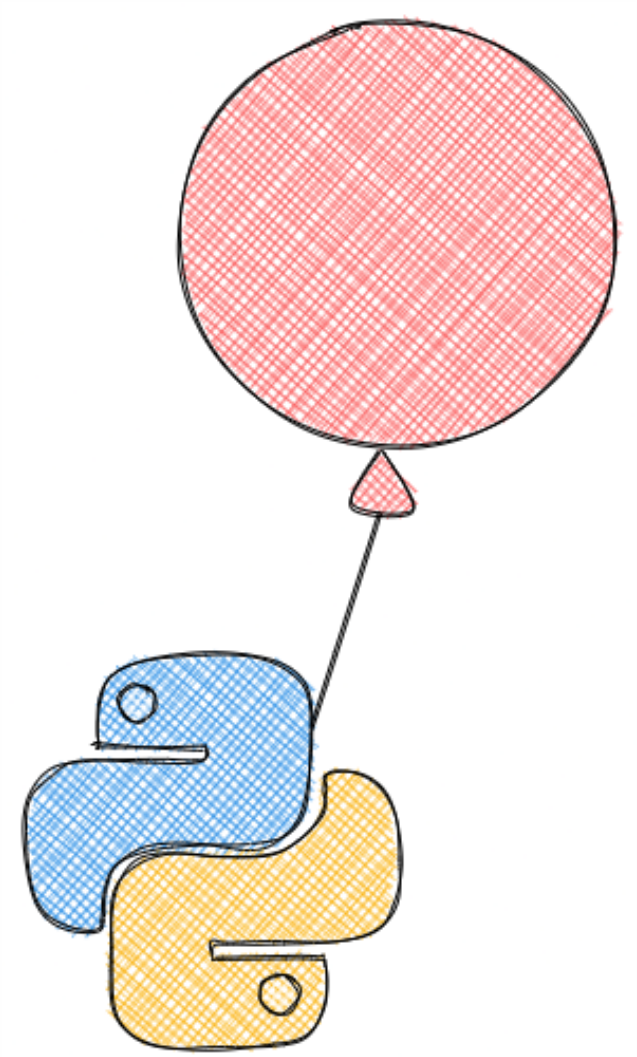
3 Techniques to Effortlessly Import and Execute Python Modules



Motivation

As a data scientist, you most likely want to share the helpful modules you created with your teammates or other users. Although your module might be useful, others will not use it if it takes them a lot of effort to access the useful functions in your module.

Thus, you want to make it easy for users to use your module. The code to import and run your module should be short. In this article, I will show you 3 ways to make it easy to import and execute your Python modules.

Import Everything

Scenario

Imagine we have a file called `utils.py` that contains all important functions and classes

```python
def add_two(num: int):
    return num + 2

def multiply_by_two(num: int):
    return num * 2

a = 5
```

In another script, we want to import everything from `utils.py` **except** the variable`a` using `from utils import *`. How can we do that?

Solution

This could easily be done by adding `__all__ = [“add_two”, “multiply_by_two”]` . Functions, classes, and packages specified in `__all__` will be imported when using `import *` .

```
__all__ = ["add_two", "multiply_by_two"]

def add_two(num: int):
    return num + 2

def multiply_by_two(num: int):
    return num * 2

a = 5
```

Now try to use `import *` in another script

```
from utils import *

num = 3
print(add_two(num))
print(multiply_by_two(num))
print(a)
```

```
5
6
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(a)
NameError: name 'a' is not defined
```

Cool! The error shows us that only `add_two` and `multiply_by_two` from `utils.py` are imported while `a` is not imported.

Combine Multiple Files

Scenario

Imagine what the structure of the files in our directory looks like below:

```
.
├── data_modules
│   ├── load_data.py
│   └── process_data.py
└── main.py
```

`load_data.py` looks like this:

```
class DataLoader:
    def __init__(self, data_dir: str):
        self.data_dir = data_dir

    def load_data(self):
        print(f"Loading data from
{self.data_dir}")
```

and `process_data.py` looks like this:

```
class DataProcessor:
    def __init__(self, data_name: str):
        self.data_name = data_name

    def process_data(self):
        print(f"Processing {self.data_name}")
```

To use classes from 2 different files, we need to import each class from each file.

```
from data_modules.load_data import DataLoader
from data_modules.process_data import
DataProcessor

data_loader = DataLoader('data/')
data_loader.load_data()

data_processor = DataProcessor('data1')
data_processor.process_data()
```

This method is fine, but it is redundant to use 2 import statements. Is there a way that we can turn two import statements into one import statement like below?

```
from data_modules import DataLoader,
DataProcessor
```

Solution

This can be easily solved with `__init__.py` file. Insert `__init__.py` file under the`data_modules` directory:

```
touch data_modules/__init__.py
```

Then insert the import statements mentioned earlier to the`__init__.py` file:

```
from .load_data import DataLoader
from .process_data import DataProcessor
```

We use one `.` here because `load_data.py` is in the same directory as `__init__.py` .

Now let's try to import `DataLoader` and `DataProcessor` from `data_modules`

```
from data_modules import DataLoader,
DataProcessor

data_loader = DataLoader('data/')
data_loader.load_data()

data_processor = DataProcessor('data1')
data_processor.process_data()
```

```
Loading data from data/
Processing data1
```

Nice! It works!

Run a Directory like a Main Script

Scenario

Our directory looks like this:

```
.
└── data_modules
    ├── __init__.py
    ├── load_data.py
    ├── main.py
    └── process_data.py
```

Instead of running

```
$ python data_modules/main.py
```

we might want to make it simpler for our users or teammates to run the code `main.py` file by simply running the parent directory:

```
$ python data_modules
```

This is better than running `python data_modules/main.py` because it is shorter, and users don't need to know what files are in `data_modules` .

How can we do that?

Solution

This is when `__main__.py` comes in handy. Simply change the script you want to run when running the top-level directory to `__main__.py`. In our example, `main.py` will become `__main__.py` .

```
# Rename main.py to __main__.py
$ mv data_modules/main.py
data_modules/__main__.py
```

Our new directory will look like this

```
.
└── data_modules
    ├── __init__.py
    ├── load_data.py
    ├── __main__.py
    └── process_data.py
```

Now run

```
$ python data_modules
Loading data from data/
Processing data1
```

And it works like a charm!

Reference

Beazley, D. M., & Jones, B. K. (2014). *Python cookbook*. Beijing ; Cambridge ; Farnham ; Köln ; Sebastopol ; Tokyo: O'Reilly.