Python Packages: Structure Code By Bundling Your Modules

August 14, 2022

We use Python packages to structure and organize our code. When talking about Python packages, people generally mean one of the following:

1. Packages installable with tools like pip and pipenv are often distributed through the Python Package Index.
2. Packages in your code base are used to structure and organize your code.

If you’re looking for how to install packages, you should read the article on installing packages with pip install instead. In addition, I can also recommend the article on virtual environments.

This article is about **creating your own packages and modules**. We’ll look at what packages are, how they are structured, and how to create a Python package. You’ll also discover how packages and modules work together to organize and structure your codebase.

If you are unfamiliar with modules, read my article on Python modules first and then come back here. These two subjects are strongly related to each other.

Table of Contents [hide]

Beginners Python Course (2023)

Modules, Packages, And Virtual Environments (2023)

What are Python packages?

A Python package is a directory that contains zero or more Python modules. A Python package can contain sub-packages, which are also directories containing modules. Each package always contains a special file named `__init__.py`. You’ll learn exactly what this mysterious file is for and how you can use it to make your package easier to import from.

Structure of a Python package

So a Python package is a folder that contains Python modules and an `__init__.py` file. The structure of a simple Python package with two modules is as follows:

```
.
└── package_name
    ├── __init__.py
    ├── module1.py
    └── module2.py
```

As mentioned, packages can contain sub-packages. We can use sub-packages to organize our code further. I'll show you how to do that in more detail in one of the sections below. Let's first take a look at the structure of a package with sub-packages:

```
.
└── package_name
    ├── __init__.py
    ├── subpackage1
        ├── __init__.py
        ├── module1.py
    └── subpackage2
        ├── __init__.py
        ├── module2.py
```

As you can see, packages are hierarchical, just like directories.

What is __init__.py in a Python package?

The `__init__.py` file is a special file that is always executed when the package is imported. When importing the package from above with `import package_name`, the `__init__.py` file is executed.

When importing the *nested* package from above, with `import package_name.subpackage1`, the `__init__.py` file of both `package_name` and `subpackage1` are executed. The order is as follows:

1. first the `__init__.py` file of `package_name` is executed,
2. then the `__init__.py` file of `subpackage1`.

I've added simple print statements to all the `__init__.py` files to demonstrate. We can create a `main.py` file in the `package_name` folder with the following contents:

```python
import package_name.subpackage1
```

If we run this program, the output will be:

```
$ python3 main.py
Hello from package_name
Hello from subpackage1
```

The print statements in both `__init__.py` files are executed due to the import of our sub-package.

Organize your code in Python packages

We now have the following tools in our belt to properly organize our Python code:

- Packages
- Sub-packages
- Modules

You should use sub-packages to group related modules together. Using sub-packages also helps you keep package and module names short and concise. They are often a good alternative when you find yourself using underscores in package names.

An example Python package

Let's create a package named `httptools` in the current directory. Our aim is that this imaginary package, one day, contains all the tools one might need to work with the HTTP protocol. We start simple, though, with just HTTP GET and POST requests and a simple HTTP server.

Our package contains two sub-packages: `client` and `server`. An initial package layout might look like this:

```
httptools/
    __init__.py
    client/
        __init__.py
        get.py
        post.py
    server/
        __init__.py
        run.py
```

Some things to notice:

- The names are short and descriptive. Because `client` and `server` are subpackages of `httptools`, it's obvious to everyone that these are an HTTP client and server. We don't need to call them `http_client` and `http_server`.
- We've grouped similar functionality in sub-packages:
  - client code in a client package,
  - and server code in a server package.
- We've grouped closely related code into modules. E.g., all we need to do HTTP get requests goes in the `get.py` module.

Extending our Python package

Now imagine that we want to jump on the async programming model. While our original server used to be synchronous, we now also offer an async variant.

There are three ways to add this to our codebase:

1. create a new package named `httptools_async`
2. create new sub-packages named `sync` and `async` in the `http` package
3. create new modules with names like `async_get` and `async_post` and async_run.

Which option would you choose?

Option 1: Create a new package

Option 1 is the most straightforward. We can create a new package named `httptools_async` and copy the contents of the `http` package into it. Next, we change all the code to be async. A user of our original package might get away with changing a single line of code: change `import http` to `import http_async`. Since asynchronous programming is a completely different paradigm, this simple swap would probably not be enough though.

Option 2: Create new sub-packages

Option two means that existing library users, even those who don't want to use async, need to change their code. It could be a good option when starting fresh, but it's not such a good option for an existing library.

Option 3: Create new modules

And then there's option three: creating new modules with async appended to the module names. Although existing users would not need to change their code, I would not recommend this. It's more logical to bundle the async modules in their own package, if only because you wouldn't have to repeat the `async` prefix/postfix all the time when doing imports. It fits in the general objective of keeping package names and module names short and concise.

Importing modules in __init__.py

Our package can be improved even more by importing the important functions inside of `__init__.py`. The `__init__.py` file is often a good place to import other modules. Let's take our `httptools.client` package from above, for example. Our get.py module provides a function `get()` that returns a response. We can import this function into the `__init__.py` file of the `client` package:

```
from httptools.client.get import get
```

This is a common pattern in Python. It's a good idea to import all the modules you need in the `__init__.py` file. If we use the import as described above, users of our package can import the get function like this:

```
from httptools.client import get
```

Depending on the situation, and also depending on taste, you could now also do this:

```
from httptools import client
client.get(...)
```

Without the import in `__init__.py`, users of our package would need to do what we did ourselves to use the get function:

```
from http.client.get import get
```

This is both more verbose and removes a bit of flexibility for us. If we decided to move the get function to some other file, it would introduce a breaking change for other parts of our code or users of our package. Because we import the function in the __init__.py of the client though, we have the flexibility to move that function around and simply change the import in __init__.py. To the users of our package, this would go unnoticed.

Absolute or relative imports

We've imported the `get()` function from the `httptools.client.get` module using an absolute import, meaning we specified the full tree leading to that function. You can also use a relative import. The advantage of relative imports is that you can import the module you want to use without knowing the full path. The package names might even change, without breaking the code. So relative imports make your code more robust to change.

A relative import in the `__init__.py` file of `httptools.client` looks like this:

```
from .get import get
```

This is also one of the cases where wildcards are useful. We can import all the elements in the `httptools.client.get` module using a wildcard:

```
from .get import *
```

This is not so bad, because we know exactly what we've put in `get.py`. In addition, we can change function names in `get.py` without needing to change import statements. This again makes the code more flexible.

Create a runnable package with __main__.py

I showed you how to create runnable modules in the lesson on modules. We can do something similar with packages by creating a file with the exact name `__main__.py`.

How to run a module in a package

To run a module from a package, we can use the following Python command:

```
python -m <module name>
```

This command can be used to run a specific module inside your package, e.g. `python -m mypackage.mymodule`, but we can also use it to run a package.

Creating a runnable package

To create a runnable package, it needs a file name `__main__.py` in its root folder. This file can import and bootstrap anything you like. Note that we don't need to add the `if __name__=='__main__'` check: we can be 100% sure the name of this module is, in fact, `__main__` since we named the file that way.

If your module is called `mymodule`, you can now run it with the following command:

```
python -m mymodule
```

Modules vs. Packages

Let’s conclude by summarizing the information about Python packages and what we learned in the previous article on Python modules. A question that often pops up is this: what’s the difference between a module and a package?

- A module is always a single file, while a package can contain many modules.
- A module is used to bundle Python functions, classes, constants, and anything else that you want to make reusable. A package, in turn, bundles modules.
- Modules can live on their own and don’t need to be part of a package, while a package needs modules to be useful.
- Together, packages and modules form a powerful way of organizing our code.

‹ Previous: Python Modules

Next: Python Data Types