How to create Python Modules, the complete tutorial

Alessandro Maggio(https://www.ictshore.com/author/alessandro-maggio/) June 7, 2018

Share This Post

Python allows you to create complex applications. If you plan to do that, you are going to write a lot of code. However, it is not practical to have all the code in a single file, it can become a mess quickly. Fortunately, Python offers an awesome way to segment your code in different files, following a hierarchical model. We are talking about Python Modules. In this article, we will see what they are, how we can use them, and how we can create our own modules.

Introducing Python Modules

In this article about python classes (https://www.ictshore.com/python/python-classes-tutorial/), we explained that a class is a collection of code that represents an entity. A module is somewhat similar because it is a collection of python code as well. However, the code in a Python module does not necessarily represent an entity. Instead, we can say the following about a module:

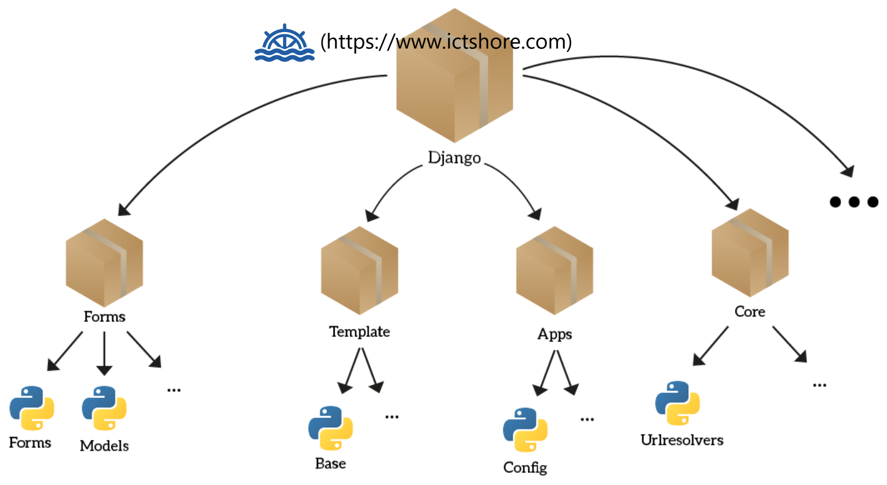
> A module is a collection of code that works together toward the same goal.

This definition may seem abstract at a first glance, but it isn't. If you are following our course, you will remember when we worked with CSV files (https://www.ictshore.com/python/files-in-python-beginners/). In that tutorial, we showed the `csv` module. That module allows you to interface with a CSV file, taking care of everything.

Remember, **a Python module can contain everything**. It can contain classes, as well as standalone functions. It may even contain static assets of a different nature, like images, although most modules are simply python code.

The tree-like structure of Python modules

In the last paragraph, we said that a Python module can contain everything. We can extend this statement and say that a Python module can contain other Python modules as well. This results in a structure that resembles a tree. For example, django (https://docs.djangoproject.com/en/2.0/) – a complex module, will look like in the following picture.



Example structure of python modules on django, a large framework. Each box represent a module, each Python icon represents a file.

Here we can see that the `django` module contains several modules, like `forms`, `template` and so on. If we look at the forms module, we can see that it contains another module named `forms`, one named `models` and other modules as well, omitted in the picture. You will need to know a little bit about the structure of the modules you are working with.

Using Python Modules

You guessed it, there is a Python module for pretty much everything. In fact, chances are that a problem similar to the one you are trying to solve is already solved with an existing Python module. Thus, we want to use Python modules from other people. Fortunately, this is extremely simple.

To use a Python module in your script, you need to use the `from` and `import` statements. Remember this:

- The `from` statement allows you to locate a position in the tree structure of the module.
- The `import` statement tells you which file of the module to import (because, as we will see, a module typically contains several files)

So, if we want to use the `loader` file from the `template` module of Django, we will use the following snippet.

```
from django.template import loader
```

Single-file modules

Although rare, some python modules are made of a single file. If that's the case, you can omit the `from` statement, as we did for the `csv` module.

```
import csv
```

Non-default Python modules

Python ships with several pre-defined modules already installed. However, not all Python Modules come with Python. You have to install the ones you need manually, as there are literally too many modules to have them all. Installing a module that was officially published is easy, you just need to use pip. This is a utility that comes with python and should be already available on your PC. If not, you can download from the web the `get-pip.py` utility that will install it for you.

Once you have pip, simply use it with the keyword *"install"* and the name of the module you want to install. Let's say we want to install `django`, we can simply use this snippet in the prompt.

```
pip install django
```

You can use pip in a more advanced way. For example, you can specify the desired version, set proxy settings, and so on. However, for most users, this usage is more than enough. As soon as you install a Python module this way, it will be available in your Python environment. All your applications will be able to import it.

Creating your own Python Modules

What's cool about Python modules

Python modules are popular for a reason: they allow your code to scale. In fact, you can have a complex application that deals with a lot of problems. With modules, you can segment your application into smaller code chucks. Each code chuck will deal with a specific issue so that instead of a big application you will be working on several small applications.

This makes **development easier** because you don't need to think about the whole picture all the time. It also creates **flexibility**, particularly if you are working in a team with multiple developers. In fact, you can have each developer focus on a different module.

If the code chucks start to be too big, you can *segment them again* into modules. This way you can keep your application manageable.

On top of that, modules are easy to export and install. You can publish your module so that other people will download them and install them.

Creating python modules 101

The simplest python module you can create is just a python file. Select a folder where you will be experimenting with modules. Then, create two files in it: `fileA.py` and `fileB.py`. Now, we can add a sample function inside `fileA`, which assumes the following content.

```python
def sample_function():
    print("Hurray!")
```

At this point, we can simply import *fileA* into *fileB* and use the sample function. You can write the following code into *fileB* and try to execute it.

```python
import fileA

fileA.sample_function()
```

Since we are just using the `sample_function()` from fileA, we can also decide to import only that, instead of the entire file. To do it, we need to use the `from` statement.

```python
from fileA import sample_function

sample_function()
```

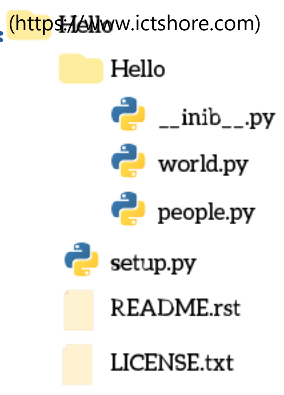
In both cases, we are using fileA module. However, this kind of module is too simple to scale to a real application. You cannot export it or install it. Furthermore, this works as long as *fileA* and *fileB* remain in the same folder. We need to develop a more solid structure.

Installable Python modules

Python's official website offers a complete guide to packaging modules (https://packaging.python.org/tutorials/distributing-packages/). Here we will explain how to create a module that can scale (with different folders), and that you can install with `pip`.

The structure

For this tutorial, we are going to create a module named *Hello*. We start by creating a folder with that name, and to structure it like the one below.



Structuring a project to be a python module will result in this type of layout.

As you can see, we created another sub-folder named hello inside our hello main folder. This sub-folder is the module itself and contains the code that we want our module to be made of. Besides the `__init__.py` file, that we will cover later, we added two files: *world* and *people*.

In our root folder, we have three files.

- `setup.py` is the most important file. This is the file that will take care of the installation of the module, running pip will execute this file.
- `README.rst` is, you guessed it, the read-me file. Someone who wants to install your package may take a look at it to see if it is what he needs.
- `LICENSE.txt` is the license file. If you plan to distribute your package to the public you must include a license, of course. You don't need that if you plan to keep your module for yourself.

The __init__ file

In our module, we have the `__init__.py` file. For the majority of modules, it can be an empty file. However, it must exist. This tells Python that this folder (`hello/hello`) is a module, and should be treated like one.

Writing some code

Once you laid out the *scaffolding*, the next step is to actually create the python module. This is an example, and we focus on creating the module rather than creating a useful module. Thus, we can write the following code in our `world.py` file.

```
class World:
    def __init__(self, world_name):
        self.name = world_name

    def hello(self):
        print("Hello, " + self.name + "!")
```

And we can add this function to our `people.py` file.

```
def say_hello(people):
    for person in people:
        print("Hello, " + person + "!")
```

Prepare the setup

Now that we have a module that actually does something, we can prepare its setup. Open the `setup.py` file, and add the following line.

```
from setuptools import setup
```

In this file we are going to need this setup utility, it is what we are going to use to coordinate the installation of the plugin. Now that we have it, we can call it – it's a function.

```
setup(name='hello',
      version='0.1',
      description='A plugin to say hello in several ways',
      url='http://ictshore.com/',
      author='Your Name',
      author_email='your.name@example.com',
      license='MIT',
      packages=['hello
      zip_safe=False)
```

As you can see, this function can accept several parameters. Here we compiled some of them that you should remember:

- `name` is the name of the plugin
- `version` is the version of your plugin
- As `description`, write a simple sentence that describes your script
- The `url` is where you can find information about the plugin, many opt for services like GitHub
- `author` is exactly you, and `author_email` is your email
- `license` is the license under which you are distributing the package
- `packages` is the list of packages contained in this module, since we just have one module we only include hello

Installing your new Python Module

At this point, our hello module is ready to be installed with pip. However, we can't simply write `pip install hello`, because this module is not registered in the public Python database. Fortunately, pip offers a way to install modules from local files, instead of contacting the public database.

To install the module locally, navigate with the prompt to the folder where you have `setup.py`. To do that, you can use the cd command on Linux and Windows. Once there, you can use a dot to tell pip *"hey, install from this very folder"*.

```
pip install .
```

This is fine but has the disadvantage to consider. If you are still testing and developing your module, you are going to change and alter its codebase. However, you are going to modify only the source, not the compressed and installed module. If you want to reflect any change in the source immediately to anywhere the module is used, you need to use `-e`. This is known as a **symlink**.

```
pip install -e .
```

Of course, you can do that only because you are on the same system where you are developing the source.

Publishing your module

Have a good module you want to share for free? Publish it on python.org (https://www.python.org/)! Doing that is simple, instead of using `pip install` use the following snippet in your prompt.

```
python setup.py register
```

This will make you create an account (if you don't have one) and register your plugin on the public repository. As Simple as that.

Wrapping it up

Now you know everything you need to be productive with Python modules. You know how to import them, how they are structured, and how to create your own. With this knowledge, you will be developing big and scalable applications before you know it. Here we have the key elements you must remember.

- To include a module in your script, use `from ... import`. The keyword `from` locates the folder or file where the module is, the `import` keyword imports the file or function/class you want to include.
- To include a module, it must be installed. Some modules comes already installed with Python, for all the others you need to use `pip install`.
- To create a plugin, you need to create two folders with the same name. In the sub-folder, put your module and an empty `__init__.py` file. In the outer folder, create your `setup.py` file.
- To install a plugin locally, and reflect any changes immediately, use `pip install -e .` utility.

So, how is it to develop your own module? Do you feel a wide range of possibilities opening? Let me know what you think about modules in the comments!

Never Stop Learning…

The Best Guide to Business Decision Making (and 9 tools to use) (https://www.ictshore.com/business/business-decision-making/)

The 5 Secret Tips to the Best Program Management Plan (https://www.ictshore.com/project-management/program-management-plan/)

(https://alessandromaggio.com)

Alessandro Maggio

(https://alessandromaggio.com)
Project manager, critical-thinker, passionate about networking & coding. I believe that time is the most precious resource we have, and that technology can help us not to waste it. I founded ICTShore.com with the same principle: I share what I learn so that you get value from it faster than I did.

Want Visibility from Tech

If you feel like sharing your knowledge, we are open to guest pos

Become an Author
(https://www.ictshore.com/guest-posti