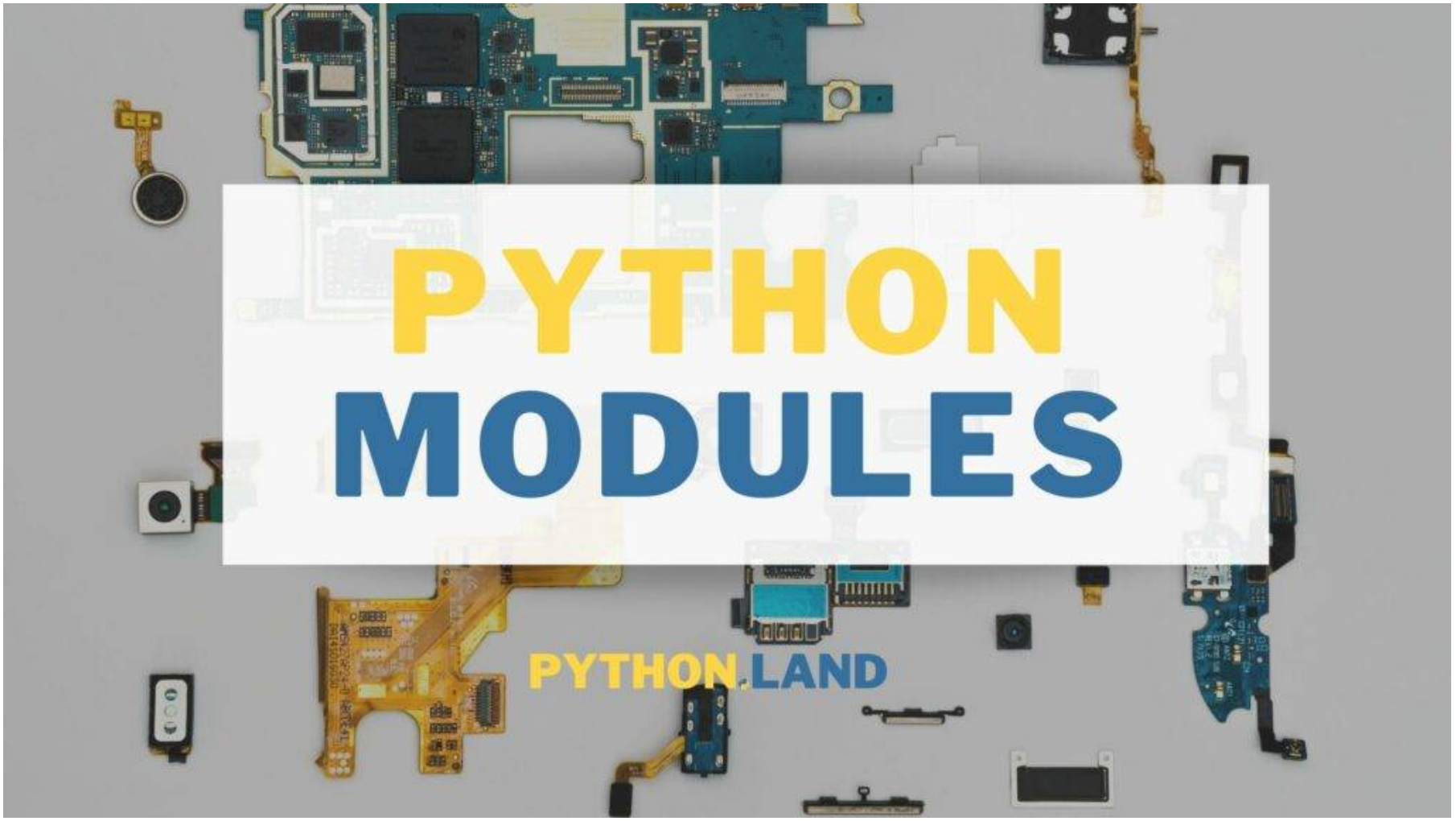
Previous: Structure Your Project

Next: Python Packages ›

Home » Structure Your Project: Modules And Packages » Python Modules: Bundle Code And Import It From Other Files



Python Modules: Bundle Code And Import It From Other Files

April 7, 2023

The Python *import* statement allows us to import a Python module. In turn, A Python module helps us organize and reuse our code. In this article, you learn about:

- How to create modules and organize your code
- The Python import statement that allows us to import (parts of) a module
- Some best practices when it comes to modules
- How to create runnable modules

In the follow-up article, you'll learn how to bundle modules in Python packages.

Table of Contents [hide]

Beginners Python Course (2023) **Modules, Packages, And Virtual Environments (2023)**

What is a module?

If you create a single Python file to perform some tasks, that's called a script. If you create a Python file to store functions, classes, and other definitions, that's called a module. We use modules by importing from them using the Python `import` statement.

So a module is a file that contains Python code ending with the `.py` extension. In other words: any Python file is also a module. The name of a module is the same as the file name, excluding the extension. For example, if you create a file named `mymodule.py`, the name of your module is `mymodule`.

Why do you need Python modules?

Modules have a lot to offer:

- They allow us to organize code. For example, you could store all functions and data related to math in a module called `math.py`. This is exactly what Python does in the math module!
- We can reuse code more easily. Instead of copying the same functions from one project to the next, we can now use the module instead. And if we properly package the module, we can even share it with others on the Python package index.

Python import

You can import a Python module by using Python's import keyword. We usually place module imports at the top of a script, but we don't have to. E.g., sometimes you want to dynamically decide if you need to import a module, especially when that module takes up a lot of memory or other resources.

Creating a module

Let's create a simple module called `mymodule.py`. All it contains is the following function:

```
def my_function():
    print('Hello world!')
```

We can import this module in another script, but you must ensure the two files are in the same directory. To import the module, use the following command:

```
import mymodule
```

This imports the entire module and makes it available to the rest of the program under the name `mymodule`. After importing the module with `import`, we can use the function by using the following command:

```
mymodule.my_function()
```

To see this in action, you can run and play around with the following code crumb:

Python import and use a module

Importing specific parts of a Python module

We can also import specific parts of a module, like a function, variable, or class. To do so, use the following syntax:

```
from mymodule import my_function
```

The `my_function` function is now available to the rest of the program under the name `my_function`, as if it was defined inside the file itself. So we can use it like this:

```
my_function()
```

Similarly, we can import multiple elements from a Python module. To do so, use the following syntax:

```
from mymodule import my_function, my_variable
```

It's up to you when to do what. Sometimes, you need a function so often that referencing the entire module each time looks messy. Other times, it's better to import the module name because you need a lot of different functions from that same module. Another consideration might be that it's clearer to the readers of your code where a specific function, variable, or class is from when you reference it by using the entire module name.

Wildcard imports (are bad practice)

When a module has a lot of functions and other elements you want to import, it can be tempting to import them all at once with a special `from module import *` syntax. To import everything from a module, we can use the following syntax:

```
from mymodule import *
```

You should avoid using the * syntax in most cases because it can lead to unexpected results, especially when using third-party modules you have no control over. After all, you don't know what's inside a third-party module or what will be inside of it in the future. By importing everything, you 'pollute' your namespace with unnecessary variables, classes, and functions. You might even overwrite some existing definitions without noticing. My advice: only import specific elements to stay in control of your namespace.

Import aliases

Sometimes Python module names are not what you like them to be. Perhaps you need to type the name a lot, like in a Jupyter Notebook. In this case, you can use an alias to shorten the module name. For example, if you have a module named `mymodule.py`, you can import it as `m`:

```
import mymodule as m
```

Some common aliases that you will encounter, especially in fields like data science, are:

- `import numpy as np`
- `import pandas as pd`
- `import matplotlib.pyplot as plt`

- `import seaborn as sns`

Valid Python module names

There are some restrictions on the names of Python modules because we must be able to import them from Python code and use/reference them with a regular variable. As a result, a valid Python module name has a lot in common with regular variable names.

A module name:

- Is a string of letters, digits
- Can contain underscores if it helps with readability
- It cannot start with a digit
- It does not contain spaces or other whitespace characters
- Cannot contain any of the following additional characters: \ / : * ? ” < > | –

In addition to these rules, there are some best practices to follow when naming modules. Notable:

- Use lowercase letters for the module name.
- Make sure names are short and descriptive.

Common mistakes

A common mistake is to use reserved words or existing module names for your module. I’ve done so myself, e.g. by creating a module called `http` while Python already has a module with this name.

Another common cause of hard-to-debug errors is naming your module the same as a directory.

How to create runnable Python modules

Inside a module, you can detect if it is used as a module or a script. This is useful because it allows you to both run a module on its own and import it at the same time from other code as well.

The `If __name__ == '__main__'` check

You might have seen it before, and you might have wondered what it does. Why do Python programmers include this check in their script so often? It has everything to do with modules and using modules as scripts.

When we import a module, its name (stored in the variable `__name__`) is equal to the module name. When a script is executed, its name is always set to the string `__main__`. Hence, if we check for this name, we find out if we are running as a script or if we're included somewhere as a module.

I created a simple module that does this check in the following code crumb. Because I didn't import this file but run it directly, the script called the greeter function and we see something printed to our screen. However, when we import this module, the script won't call the greeter function:

A runnable module that checks if it is used as a module or a script

By executing code only when running in 'scripting mode', we effectively create files that can act as both a script and as a module that we can import.

Classes versus modules

You might wonder how modules compare to classes. They both group functionality, after all. There are significant differences, though, and there are good reasons for Python to have both modules and classes.

A module groups similar functionality or related functionality into one place. A module, for example, might contain classes, functions, and constants that we use to connect to a particular type of database.

A class models your problem domain. It provides a blueprint for one or more objects. E.g., a student class holds student data and related functions that work on that data. There can be many students, and hence there can be many student objects simultaneously. For more on this, read the chapter on classes and objects.

Further resources

The python.org page, listing the Python standard library, gives a nice overview of built-in modules.

Conclusion

You've learned how to bundle code into a module, helping you structure your project. In the process, your project will become more maintainable. Modules are just one piece of the puzzle. In the next article on Python packages, you'll learn how modules and packages work together to structure your project further.

‹ Previous: Structure Your Project

Next: Python Packages