CONTENTS

Prerequisites

Writing and Importing Modules

Accessing Modules from Another Directory

Conclusion

Tutorial Series: How To Code in Python

22/36 How To Write Modules in ...

23/36 How To Write Conditional...

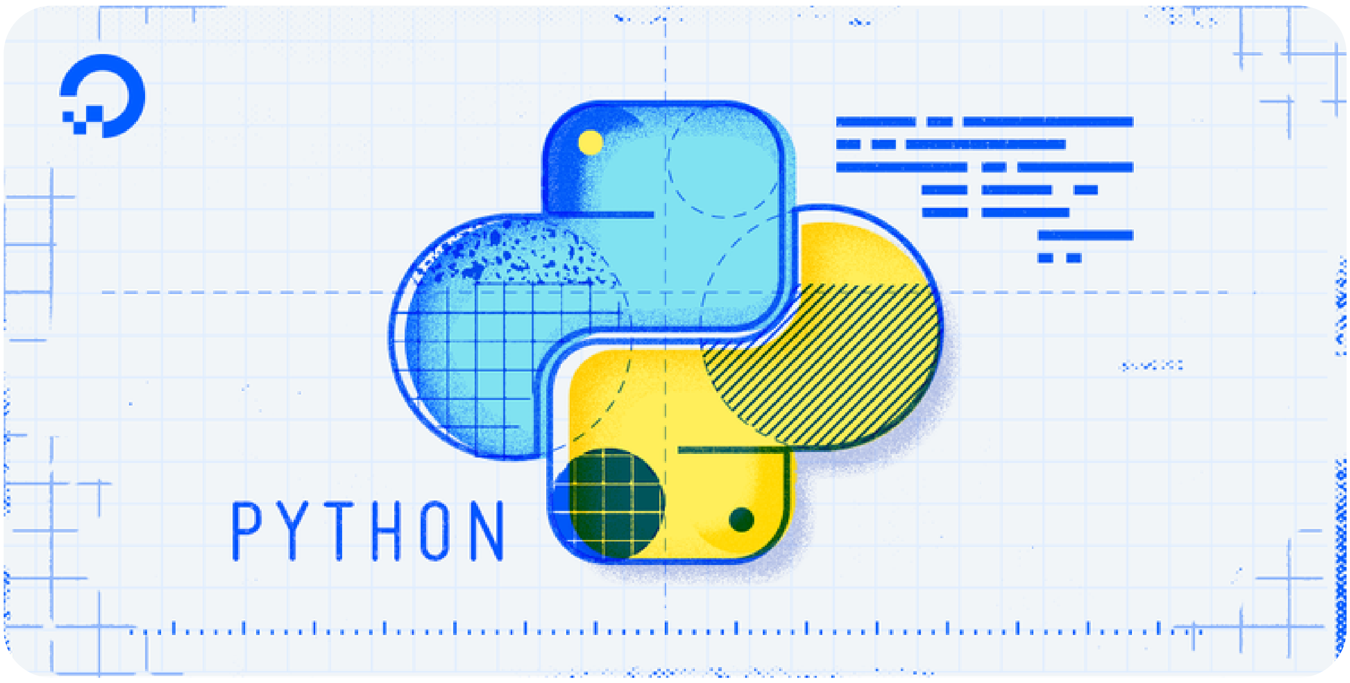
// Tutorial //

How To Write Modules in Python 3

Updated on August 20, 2021

Python Development

By Lisa Tagliaferri



Introduction

Python **modules** are `.py` files that consist of Python code. Any Python file can be referenced as a module.

Some modules are available through the Python Standard Library and are therefore installed with your Python installation. Others can be installed with Python's package manager `pip`. Additionally, you can create your own Python modules since modules are comprised of Python `.py` files.

This tutorial will guide you through writing Python modules for use within other programming files.

Prerequisites

You should have Python 3 installed and a programming environment set up on your computer or server. If you don't have a programming environment set up, you can refer to the installation and setup guides for a local programming environment or for a programming environment on your server appropriate for your operating system (Ubuntu, CentOS, Debian, etc.)

Writing and Importing Modules

Writing a module is like writing any other Python file. Modules can contain definitions of functions, classes, and variables that can then be utilized in other Python programs.

Info: To follow along with the example code in this tutorial, open a Python interactive shell on your local system by running the `python3` command. Then you can copy, paste, or edit the examples by adding them after the `>>>` prompt.

From our Python 3 local programming environment or server-based programming environment, let's start by creating a file `hello.py` that we'll later import into another file.

To begin, we'll create a function that prints `Hello, World!`:

hello.py

```
# Define a function
def world():
    print("Hello, World!")
```

Copy

If we run the program on the command line with `python hello.py` nothing will happen since we have not told the program to do anything.

Let's create a second file **in the same directory** called `main_program.py` so that we can import the module we just created, and then call the function. This file needs to be in the same directory so that Python knows where to find the module since it's not a built-in module.

main_program.py

```
# Import hello module
import hello


# Call function
hello.world()
```

Copy

Because we are importing a module, we need to call the function by referencing the module name in dot notation.

We could instead import the module as `from hello import world` and call the function directly as `world()`. You can learn more about this method by reading how to using
... [illegible] when importing modules.

Now, w[illegible]n run the program on the command line:

```
$ python main_program.py
```

Copy

When we do, we'll receive the following output:

```
Output
Hello, World!
```

To see how we can use variables in a module, let's add a variable definition in our `hello.py` file:

hello.py

```
# Define a function
def world():
    print("Hello, World!")

# Define a variable
shark = "Sammy"
```

Copy

Next, we'll call the variable in a `print()` function within our `main_program.py` file:

main_program.py

```
# Import hello module
import hello


# Call function
hello.world()

# Print variable
print(hello.shark)
```

Copy

Once we run the program again, we'll receive the following output:

```
Output
```

```
Hello, World!
Sammy
```

Finally, let's also define a class in the `hello.py` file. We'll create the class `Octopus` with `name` and `color` attributes and a function that will print out the attributes when called.

hello.py

Copy

```
# Define a function
def world():
    print("Hello, World!")

# Define a variable
shark = "Sammy"


# Define a class
class Octopus:
    def __init__(self, name, color):
        self.color = color
        self.name = name

    def tell_me_about_the_octopus(self):
        print("This octopus is " + self.color + ".")
        print(self.name + " is the octopus's name.")
```

We'll now add the class to the end of our `main_program.py` file:

main_program.py

Copy

```
# Import hello module
import hello


# Call function
hello.world()

# Print variable
print(hello.shark)

# [illegible]class
```

```
jesse = hello.Octopus("Jesse", "orange")
jesse.tell_me_about_the_octopus()
```

Once we have called the Octopus class with `hello.Octopus()`, we can access the functions and attributes of the class within the `main_program.py` file's namespace. This lets us write `jesse.tell_me_about_the_octopus()` on the last line without invoking `hello`

```
Hello, World!
Sammy
This octopus is orange.
Jesse is the octopus's name.
```

It is important to keep in mind that though modules are often definitions, they can also implement code. To see how this works, let's rewrite our `hello.py` file so that it implements the `world()` function:

hello.py

```
# Define a function
def world():
    print("Hello, World!")

# Call function within module
world()
```

Copy

We have also deleted the other definitions in the file.

Now, in our `main_program.py` file, we'll delete every line except for the import statement:

main_program.py

```
# Import hello module
import hello
```

Copy

When we run `main_program.py` we'll receive the following output:

```
Output
Hello, World!
```

This is because the `hello` module implemented the `world()` function which is then passed to `main_program.py` and executes when `main_program.py` runs.

A module is a Python program file composed of definitions or code that you can leverage in other Python program files.

Accessing Modules from Another Directory

Modules may be useful for more than one programming project, and in that case it makes less sense to keep a module in a particular directory that's tied to a specific project.

If you want to use a Python module from a location other than the same directory where your main program is, you have a few options.

Appending Paths

One option is to invoke the path of the module via the programming files that use that module. This should be considered more of a temporary solution that can be done during the development process as it does not make the module available system-wide.

To append the path of a module to another programming file, you'll start by importing the `sys` module alongside any other modules you wish to use in your main program file.

The `sys` module is part of the Python Standard Library and provides system-specific parameters and functions that you can use in your program to set the path of the module you wish to implement.

For example, let's say we moved the `hello.py` file and it is now on the path `/usr/ sammy /` while the `main_program.py` file is in another directory.

In our `main_program.py` file, we can still import the `hello` module by importing the `sys` module and then appending `/usr/ sammy /` to the path that Python checks for files.

main_program.py

```
import sys
sys.path.append('/usr/sammy/')
```

Copy

```
import hello
...
```

As long as you correctly set the path for the `hello.py` file, you'll be able to run the `main_program.py` file without any errors and receive the same output as above when `hello.py` was in the same directory.

Adding the Module to the Python Path

A second option that you have is to add the module to the path where Python checks for modules and packages. This is a more permanent solution that makes the module available environment-wide or system-wide, making this method more portable.

To find out what path Python checks, run the Python interpreter from your programming environment:

```
$ python3
```

Copy

Next, import the `sys` module:

```
>>> import sys
```

Copy

Then have Python print out the system path:

```
>>> print(sys.path)
```

Copy

Here, you'll receive some output with at least one system path. If you're in a programming environment, you may receive several. You'll want to look for the one that is in the environment you're currently using, but you may also want to add the module to your main system Python path. What you're looking for will be similar to this:

```
Output
'/usr/ sammy / my_env /lib/python3.5/site-packages'
```

Now you can move your `hello.py` file into that directory. Once that is complete, you can import the `hello` module as usual:

main_program.py

```
import hello
...
```

When you run your program, it should complete without error.

Modifying the path of your module can ensure that you can access the module regardless of what directory you are in. This is useful especially if you have more than one project referencing a particular module.

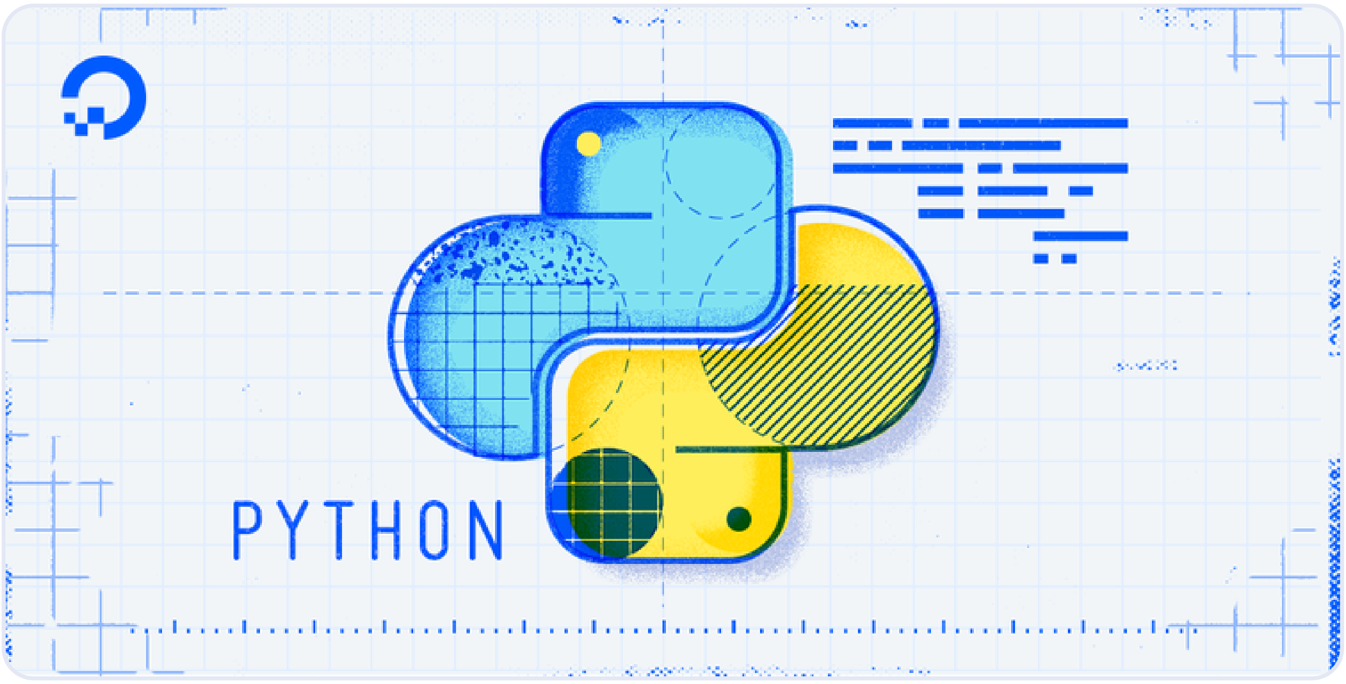
Conclusion

Writing a Python module is the same as writing any other Python `.py` file. This tutorial covered how to write definitions within a module, make use of those definitions within another Python programming file, and went over options of where to keep the module in order to access it.

You can learn more about installing and importing modules by reading How To Import Modules in Python 3.

Next in series: How To Write Conditional Statements in Python 3 →

Tutorial Series: How To Code in Python



Introduction

Python is a flexible and versatile programming language that can be leveraged for many use cases, with strengths in scripting, automation, data analysis, machine learning, and back-end development. It is a great tool for both new learners and experienced developers alike.

Prerequisites

You should have Python 3 installed and a programming environment set up on your computer or server. If you don’t have a programming environment set up, you can refer to the installation and setup guides for a local programming environment or for a programming environment on your server appropriate for your operating system (Ubuntu, CentOS, Debian, etc.)

Subscribe

Python Development

Browse Series: 36 articles

- 1/36 How To Write Your First Python 3 Program
- 2/36 How To Work with the Python Interactive Console
- 3[illegible]ow To Write Comments in Python 3

Expand to view all

About the authors

Lisa Tagliaferri Author

Comments

4 Comments

MohamedSalah225 • June 25, 2018

this is the most usefull pyhton tutorial on the internet , thank you

Reply

RSHirsch • September 22, 2019

Question: How do I design a module to work with:

```
from MODULE import OBJECT
```

I'm not sure but i THINK, that each object in a module is importable on its own, giving my main program access to the one object rather than all the objects. Is this correct?

So in your example coudl I have code that says

```
from hello import octopus
```

?

Reply

saahmathworks • September 19, 2019

Clearly explained! thank you

Reply

alekgrigorian • January 14, 2018

C[illegible]ut noticed that my .pyc for my module is for 2.7 rather than 3.5

Reply

Try DigitalOcean for free

Click below to sign up and get **$200 of credit** to try our products over 60 days!

Sign up

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

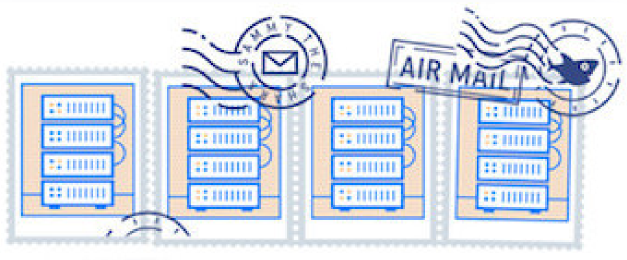
Talk to an expert →

ngratulations on unlocking the whale ambience easter egg! Click the whale button in ottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg



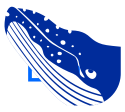
Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

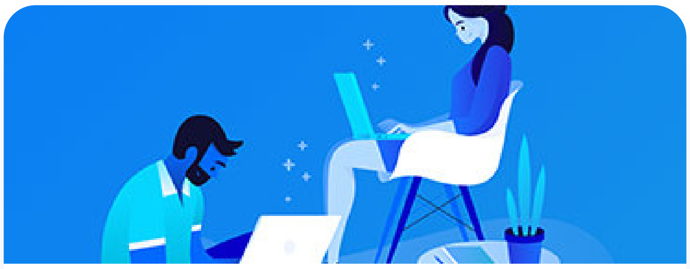
Sign up →

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.



more →



Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Featured on Community

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

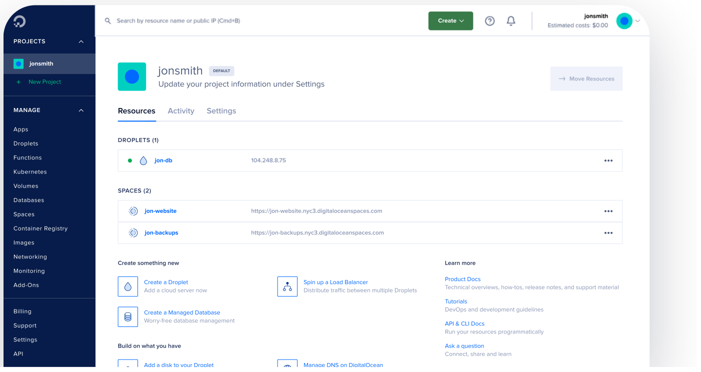
Cloudways Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn more →



Company

Products

Community

Solutions

Contact