How To Import Modules in Python 3

Introduction

The Python programming language comes with a variety of built-in functions. Among these are several common functions, including:

- `print()` which prints expressions out
- `abs()` which returns the absolute value of a number
- `int()` which converts another data type to an integer
- `len()` which returns the length of a sequence or collection

These built-in functions, however, are limited, and we can make use of modules to make more sophisticated programs.

Modules are Python `.py` files that consist of Python code. Any Python file can be referenced as a module. A Python file called `hello.py` has the module name of `hello` that can be imported into other Python files or used on the Python command line interpreter.

Modules can define functions, classes, and variables that you can reference in other Python `.py` files or via the Python command line interpreter.

In Python, modules are accessed by using the `import` statement. When you do this, you execute the code of the module, keeping the scopes of the definitions so that your current file(s) can make use of these.

When Python imports a module called `hello` for example, the interpreter will first search for a built-in module called `hello`. If a built-in module is not found, the Python interpreter will then search for a file named `hello.py` in a list of directories that it receives from the `sys.path` variable.

This tutorial will walk you through checking for and installing modules, importing modules, and aliasing modules.

Checking For and Installing Modules

There are a number of modules that are built into the **Python Standard Library**, which contains many modules that provide access to system functionality or provide standardized solutions. The Python Standard Library is part of every Python installation.

To check that these Python modules are ready to go, enter into your local Python 3 programming environment or server-based programming environment and start the Python interpreter in your command line.

From within the interpreter you can run the `import` statement to make sure that the given module is ready to be called, as in:

```
import math
```

Since `math` is a built-in module, your interpreter should complete the task with no feedback, returning to the prompt. This means you don’t need to do anything to start using the `math` module.

Let's run the `import` statement with a module that you may not have installed, like the 2D plotting library `matplotlib`:

```
import matplotlib
```

If `matplotlib` is not installed, you'll receive an error like this:

```
Output
ImportError: No module named 'matplotlib'
```

You can install `matplotlib` with `pip`.

Next, we can use `pip` to install the `matplotlib` module:

```
pip install matplotlib
```

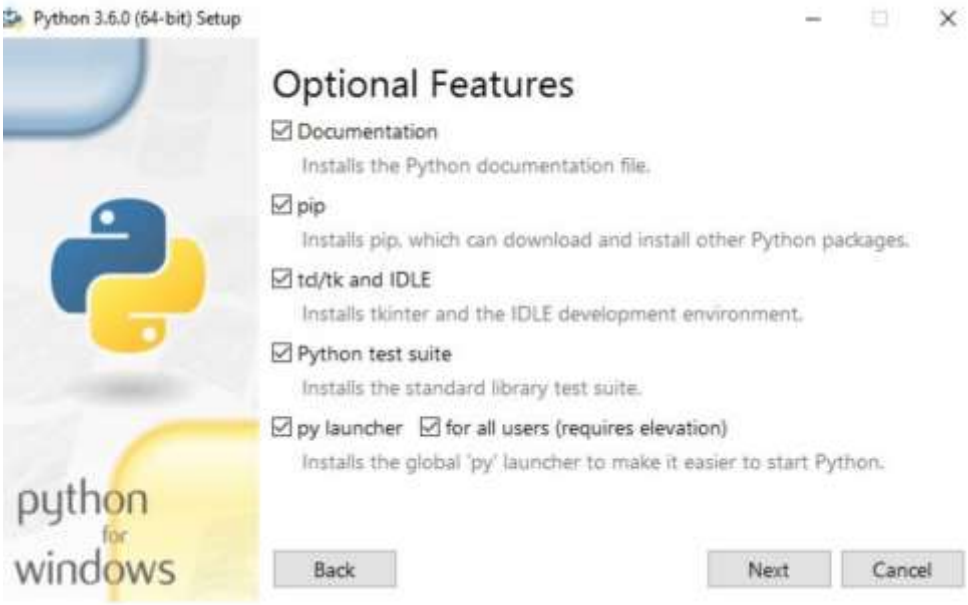
Once it is installed, you can import `matplotlib` in the Python interpreter using `import matplotlib`, and it will complete without error.

Pip Installation

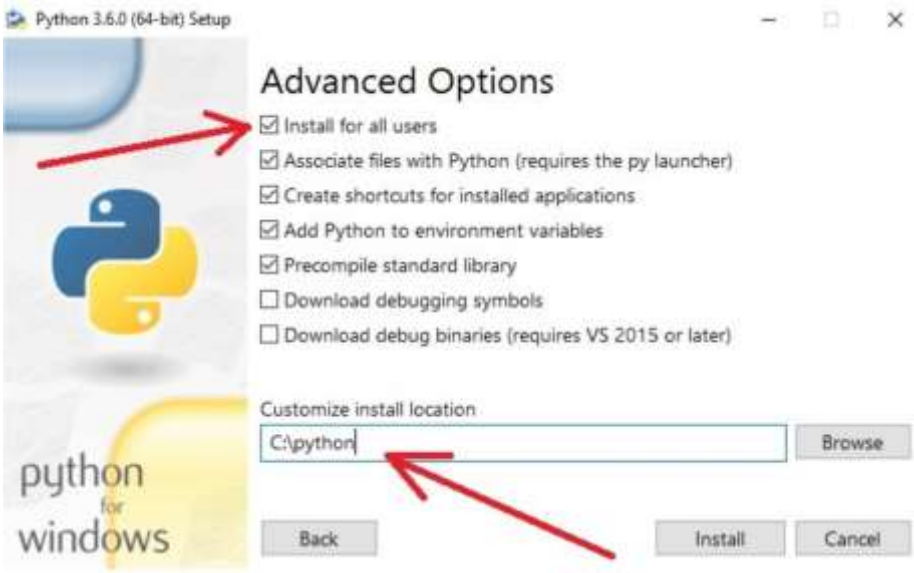
This is a advice how to install Python and pip under Windows.
It's advisable to change the long default Path,to a simper Path eg "C:\Python36"

- Download python from http://www.python.org/
  Choose a executable installer 32-bit or 64-bit.

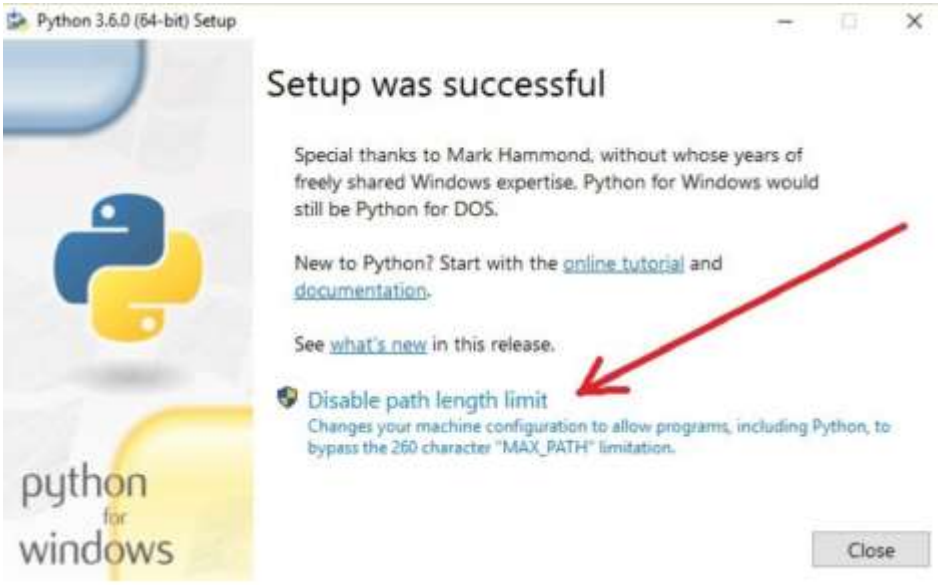
- Under Customize installation make sure that pip is marked on.



- Here also choose Path C:\Python36



- Finish install.



- Restart

- Testing that python and pip command work from cmd.
  Start cmd:

```
Microsoft Windows [Version 10.0.14393]

(c) 2016 Microsoft Corporation.

C:\Windows\System32>cd\

C:\>python

Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016,
07:18:10) [MSC v.1900 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license"
for more information.

>>> exit()

C:\>pip -V

pip 9.0.1 from c:\python36\lib\site-packages
(python 3.6)

C:\>
```

Importing Modules

To make use of the functions in a module, you'll need to import the module with an `import` statement.

An `import` statement is made up of the `import` keyword along with the name of the module.

In a Python file, this will be declared at the top of the code. So, in the Python program file `my_rand_int.py` we would import the `random` module to generate random numbers in this manner:

my_rand_int.py

```
import random
```

When we import a module, we are making it available to us in our current program as a separate namespace. This means that we will have to refer to the function in dot notation, as in `[module].[function]`.

In practice, with the example of the `random` module, this may look like a function such as:

- `random.randint()` which calls the function to return a random integer, or
- `random.randrange()` which calls the function to return a random element from a specified range.

Let's create a `for` loop to show how we will call a function of the `random` module within our `my_rand_int.py` program:

my_rand_int.py

```
import random
for i in range(10):
    print(random.randint(1, 25))
```

This small program first imports the `random` module on the first line, then moves into a `for` loop which will be working with 10 elements. Within the loop, the program will print a random integer within the range of 1 through 25 (inclusive). The integers `1` and `25` are passed to `random.randint()` as its parameters.

When we run the program, we'll receive 10 random integers as output. Because these are random you'll likely get different integers each time you run the program, but they'll look something like this:

```
Output
6
9
1
14
3
22
10
1
15
9
```

The integers should never go below 1 or above 25.

If you would like to use functions from more than one module, you can do so by adding multiple `import`statements:

my_rand_int.py

```
import random
import math
```

To make use of our additional module, we can add the constant `pi` from `math` to our program, and decrease the number of random integers printed out:

my_rand_int.py

```
import random
import math
for i in range(5):
    print(random.randint(1, 25))
print(math.pi)
```

Now, when we run our program, we'll receive output that looks like this, with an approximation of pi as our last line of output:

```
Output
18
10
7
13
10
3.141592653589793
```

The `import` statement allows you to import one or more modules into your Python program, letting you make use of the definitions constructed in those modules.

Using `from ... import`

To refer to items from a module within your program's namespace, you can use the `from` ... `import`statement. When you import modules this way, you can refer to the functions by name rather than through dot notation

In this construction, you can specify which definitions to reference directly.

Let's first look at importing one specific function, `randint()` from the `random` module:

my_rand_int.py

```
from random import randint
```

Here, we first call the `from` keyword, then `random` for the module. Next, we use the `import` keyword and call the specific function we would like to use.

Now, when we implement this function within our program, we will no longer write the function in dot notation as `random.randint()` but instead will just write `randint()`:

my_rand_int.py

```
from random import randint
for i in range(10):
    print(randint(1, 25))
```

When you run the program, you'll receive output similar to what we received earlier.

Using the `from` ... `import` construction allows us to reference the defined elements of a module within our program's namespace, letting us avoid dot notation.

Aliasing Modules

It is possible to modify the names of modules and their functions within Python by using the `as` keyword.

You may want to change a name because you have already used the same name for something else in your program, another module you have imported also uses that name, or you may want to abbreviate a longer name that you are using a lot.

The construction of this statement looks like this:

```
import [module] as [another_name]
```

Let's modify the name of the `math` module in our `my_math.py` program file. We'll change the module name of `math` to `m` in order to abbreviate it. Our modified program will look like this:

my_math.py

```
import math as m
print(m.pi)
print(m.e)
```

Within the program, we now refer to the `pi` constant as `m.pi` rather than `math.pi`.

For some modules, it is commonplace to use aliases. The `matplotlib.pyplot` module's official documentation calls for use of `plt` as an alias:

```
import matplotlib.pyplot as plt
```

This allows programmers to append the shorter word `plt` to any of the functions available within the module, as in `plt.show()`.

How To Write Modules in Python 3

Introduction

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

Writing and Importing Modules

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

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!")
```

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()
```

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()`. Now, we can run the program on the command line:

```
python main_program.py
```

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"
```

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)
```

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

```
Output
Hello, World!
Sammy
```

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()
```

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
```

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/')
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:

```
python
```

Next, import the `sys` module:

```
import sys
```

Then have Python print out the system path:

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

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.

Now you can move your `hello.py` file into one of those directories. 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.