**Python Modules**

In Python, modules refer to the Python file, which contains Python code like Python statements, classes, functions, variables, etc. A file with Python code is defined with extension.py

For example: In Test.py, where the test is the module name.

In Python, large code is divided into small modules. The benefit of modules is, it provides a way to share reusable functions.

**Table of contents**

* [Types of modules](https://pynative.com/python-modules/#h-types-of-modules)
  + [Built-in modules](https://pynative.com/python-modules/#h-built-in-modules)
  + [User-defined modules](https://pynative.com/python-modules/#h-user-defined-modules)
* [How to import modules?](https://pynative.com/python-modules/#h-how-to-import-modules)
  + [Import multiple modules](https://pynative.com/python-modules/#h-import-multiple-modules)
  + [Import only specific classes or functions from a module](https://pynative.com/python-modules/#h-import-only-specific-classes-or-functions-from-a-module)
  + [Import with renaming a module](https://pynative.com/python-modules/#h-import-with-renaming-a-module)
  + [Import all names](https://pynative.com/python-modules/#h-import-all-names)
* [Create Module](https://pynative.com/python-modules/#h-create-module)
  + [Variables in Module](https://pynative.com/python-modules/#h-variables-in-module)
* [Python Module Search Path](https://pynative.com/python-modules/#h-python-module-search-path)
* [Reloading a module](https://pynative.com/python-modules/#h-reloading-a-module)
* [The dir() function](https://pynative.com/python-modules/#h-the-dir-function)
* [Related](https://pynative.com/python-modules/#h-related)

**Types of modules**

In Python, there are two types of modules.

1. Built-in Modules
2. User-defined Modules

**Built-in modules**

Built-in modules come with default Python installation. One of Python’s most significant advantages is its rich library support that contains lots of built-in modules. Hence, it provides a lot of reusable code.

Some commonly used Python built-in modules are datetime, os, math, sys, random, etc.

**User-defined modules**

The modules which the user defines or create are called a **user-defined module**. We can create our own module, which contains classes, functions, variables, etc., as per our requirements.

**How to import modules?**

In Python, the import statement is used to import the whole module. Also, we can import specific classes and functions from a module.

For example, import module name.

When the interpreter finds an import statement, it imports the module presented in a search path. The module is loaded only once, even we import multiple times.

To import modules in Python, we use the Python import keyword. With the help of the import keyword, both the **built-in** and **user-defined** modules are imported. Let’s see an example of importing a math module.

**import** math

# use math module functions

**print**(math.sqrt(5))

# Output 2.23606797749979

**Import multiple modules**

If we want to use more than one module, then we can import multiple modules. This is the simplest form of import statement that we already used in the above example.

**Syntax of import statement:**

**import** module1[,module2[,.. moduleN]

**Example**

# Import two modules

**import** math, random

**print**(math.factorial(5))

**print**(random.randint(10, 20))

**Output**

120

18

**Import only specific classes or functions from a module**

To import particular classes or functions, we can use the from...import statement. It is an alternate way to import. Using this way, we can import individual attributes and methods directly into the program.

In this way, we are not required to use the module name. See the following example.

**Syntax of from...import statement:**

**from** <module\_name> **import** <name(s)>

**Example**

# import only factorial function from math module

**from** math **import** factorial

**print**(factorial(5))

**Output**

120

**Import with renaming a module**

If we want to use the module with a different name, we can use from..import…as statement.

It is also possible to import a particular method and use that method with a different name. It is called **aliasing**. Afterward, we can use that name in the entire program.

**Syntax of from..import ..as keyword:**

**from** <module\_name> **import** <name> **as** <alternative\_name>

**Example 1**: Import a module by renaming it

**import** random **as** rand

**print**(rand.randrange(10, 20, 2))

**Output**

16

**Example 2**: import a method by renaming it

# rename randint as random\_number

**from** random **import** randint **as** random\_number

# Gives any random number from range(10, 50)

**print**(random\_number(10, 50))

**Output**

32

**Import all names**

If we need to import all functions and attributes of a specific module, then instead of writing all function names and attribute names, we can import all using an **asterisk** \*.

**Syntax of import \* statement:**

**import** \*

**Example**

**from** math **import** \*

**print**(**pow**(4,2))

**print**(factorial(5))

**print**(pi\*3)

**print**(sqrt(100))

**Output**

16.0

120

9.42477796076938

10.0

**Create Module**

In Python, to create a module, write Python code in the file, and save that file with the.py extension. Here our module is created.

**Example**

**def** my\_func():

**print**("Learn Python with PYnative")

**Output**

Learn Python with PYnative

**Variables in Module**

In Python, the module contains Python code like classes, functions, methods, but it also has variables. A variable can list, tuple, dict, etc.

Let’s see this with an example:

First, create a Python module with the name test\_module.py and write the below code in that file.

**Example**

cities\_list = ['Mumbai', 'Delhi', 'Bangalore', 'Karnataka', 'Hyderabad']

Now, create a Python file with the name test\_file.py, write the below code and import the above module test\_module.py in that file. See the following code.

**import** test\_module

# access first city

city = test\_module.cities\_list[1]

**print**("Accessing 1st city:", city)

# Get all cities

cities = test\_module.cities\_list

**print**("Accessing All cities :", cities)

When we execute this test\_file.py, the variable of test\_module.py is accessible using the dot(.)operator.

**Output**

Accessing 1st city: Delhi

Accessing All cities : ['Mumbai', 'Delhi', 'Bangalore', 'Karnataka', 'Hyderabad']

**Python Module Search Path**

When we import any program module, the interpreter first searches for a specified name for a built-in module. If the name is not found, the interpreter searches in a list of directories given by the variable sys.path which initialized from the environment variable PYTHONPATH.

PYTHONPATH have the same syntax as the Unix shell variable PATH, list of the colon(:)-separated directory names. When a PYTHONPATH is not set, or the file is not found there, the search continues in an installation-dependent default path. It is usually /usr/local/lib/python.

**import** sys

**print**(sys.path)

**Reloading a module**

In Python, when we import a module in our program using the import statement, the module is loaded. By default, the module loaded only once, even if we import it multiple times.

Sometimes we update the loaded module with new changes, then an updated version of the module is not available to our program. In that case, we can use the reload() function to reload a module again.

First, create a Python module with the name test\_module.py and write the below code in that file.

**print**("Welcome to PYnative")

Now, create a Python file with the name, test\_file.py and write the below code in it and import the module test\_module.py. See the following code.

**import** time

**from** importlib **import** **reload**

# load 1st time

**import** test\_module

time.sleep(20)

# reload

**reload**(test\_module)

time.sleep(20)

# reload again

**reload**(test\_module)

**print**("This is test file..")

**Output**

Welcome to PYnative

Welcome to PYnative

Welcome to PYnative

This is test file..

**The dir() function**

In Python, dir() is a built-in function. This function is used to list all members of the current module. When we use this function with any object (an object can be sequence like list, tuple, set, dict or can be class, function, module, etc. ), it returns properties, attributes, and method.

For Class Objects, it returns a list of names of all the valid attributes and base attributes.

**Syntax of dir() function:**

**dir**([**object**])

**Example**

**import** math

**print**(**dir**(math))

**Output**

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

**Return value from dir()**

* When we use dir() with an object, it returns the list of the object’s attributes.
* When we use the \_\_dir\_\_() The object’s method, if that object has this method, it returns all attributes of that object. And if that object does not has \_\_dir\_\_() method, it returns all information about that object.
* If we do not pass an object to dir() it returns a list of currently available functions, methods, properties, attributes, names in the local scope.