Python Modules

Introduction

- **Python Module** is a file that **contains built-in functions**, **classes**, its and **variables**. There are many Python modules, each with its specific work.

- In this article, we will cover everything about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the **alias** to **rename** the module, etc.

What is Python Module

- A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

- Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

- To **create** a Python module, write the desired code and save that in a file with **.py extension**.

Creating a Python Module Example

Let’s create a simple calc.py in which we define two functions, one add, and another subtract.

```
1  # A simple module, calc.py
2  def add(x, y):
3      return (x+y)
4
5  def subtract(x, y):
6      return (x-y)
```

Import module in Python

- We can **import** the **functions**, and **classes defined** in a **module** to another module using the import statement in some other Python source file.

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

- Note: A search path is a list of directories that the interpreter searches for importing a module.

Importing modules in Python Example

For example, to import the module calc.py, we need to put the following command at the top of the script.

Note: This does not import the functions or classes directly instead imports the module only.

To access the functions inside the module the dot(.) operator is used.

Importing modules in Python Example

Now, we are importing the calc that we created earlier to perform add operation.

```
1  # importing  module calc.py
2  import calc
3
4  print(calc.add(10, 2))
```

```
12
```

Python Import From Module

- Python from statement lets you import specific attributes from a module without importing the module.

Import Specific Attributes from a Python module

Here, we are importing specific sqrt and factorial attributes from the math module.

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

```
4.0
720
```

Import all Names

- The * symbol used with the import statement is used to import all the names from a module to a current namespace.

```
from module_name import *
```

What does import * do in Python?

The use of * has its advantages and disadvantages.

If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the
# module math
from math import *

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

```
4.0
720
```

Locating Python Modules

- Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner –

  - First, it searches for the module in the current directory.
  - If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
  - If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules

Here, sys.path is a built-in variable within the sys module.

It contains a list of directories that the interpreter will search for the required module.

```
1  # importing sys module
2  import sys
3
4  # importing sys.path
5  print(sys.path)
```

```
['/home/nikhil/Desktop/gfg', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '', '/home/nikhil/.local/lib/python3.8/site-packages',
'/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages', '/usr/local/lib/python3.8/dist-packages/IPython/extensions', '/home/nikhil/.ipython']
```

Renaming the Python Module

We can rename the module while importing it using the keyword.

Syntax: Import Module_name as Alias_name

```
# importing sqrt() and factorial from the
# module math
import math as mt

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(mt.sqrt(16))
print(mt.factorial(6))
```

```
4.0
720
```

Python Built-in modules

There are several built-in modules in Python, which you can import whenever you like.

Examples:

math, random, datetime

```python
# importing built-in module math
import math

# using square root(sqrt) function contained
# in math module
print(math.sqrt(25))

# using pi function contained in math module
print(math.pi)

# 2 radians = 114.59 degrees
print(math.degrees(2))

# 60 degrees = 1.04 radians
print(math.radians(60))

# Sine of 2 radians
print(math.sin(2))

# Cosine of 0.5 radians
print(math.cos(0.5))

# Tangent of 0.23 radians
print(math.tan(0.23))

# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))
```

```
5.0
3.14159265359
114.591559026
1.0471975512
0.909297426826
0.87758256189
0.234143362351
24
```

```python
# importing built in module random
import random

# printing random integer between 0 and 5
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())

# random number between 0 and 100
print(random.random() * 100)

List = [1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing
# a random element from a set such as a list
print(random.choice(List))
```

```
3
0.401533172951
88.4917616788
True
```

```python
# importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the
# Unix Epoch, January 1st 1970
print(time.time())

# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

```
1461425771.87
```

Python Main Function

Python Main Function

- **Main** function is like the **entry point of a program**.
- However, **Python interpreter runs the code right from the first line**.
- The execution of the code **starts from the starting line** and **goes line by line**.
- It **does not matter** where the **main function** is **present**, or it is present **or not**.

Setting a Main Function in Python

Since there is no main() function in Python, when the command to run a Python program is given to the interpreter, the code that is at level 0 indentation is to be executed.

However, before doing that, it will define a few special variables. __name__ is one such special variable.

If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value __main__.

If this file is being imported from another module, __name__ will be set to the module's name.

__name__ is a built-in variable which evaluates to the name of the current module.

```
# Python program to demonstrate
# main() function

print("Hello")

# Defining main function
def main():
    print("hey there")


# Using the special variable
# __name__
if __name__=="__main__":
    main()
```

```
Hello
hey there
```

When the program is executed, the interpreter declares the initial value of name as "main".
When the interpreter reaches the if statement it checks for the value of name and when the value of if is true it runs the main function else the main function is not executed.

Main Function as a Module

- Now when we import a Python script as module the **__name__** variable gets the value same as the name of the python script imported.

Main function as Module Example

Let's consider there are two Files(File1.py and File2.py).

File1 is as follow.

```python
# File1.py

print("File1 __name__ = %s" %__name__)

if __name__ == "__main__":
    print("File1 is being run directly")
else:
    print("File1 is being imported")
```

```
File1 __name__ = __main__
File1 is being run directly
```

Main function as Module Example

Now, when the **File1.py is imported** into **File2.py**, the value of **__name__ changes**.

```python
# File1.py

print("File1 __name__ = %s" %__name__)

if __name__ == "__main__":
    print("File1 is being run directly")
else:
    print("File1 is being imported")
```

```
File1 __name__ = __main__
File1 is being run directly
```

```python
# File2.py

import File1

print("File2 __name__ = %s" %__name__)

if __name__ == "__main__":
    print("File2 is being run directly")
else:
    print("File2 is being imported")
```

```
File1 __name__ = File1
File1 is being imported
File2 __name__ = __main__
File2 is being run directly
```

To understand

- When File1.py is run **directly**, the **interpreter sets the __name__ variable as __main__** and when it is **run through File2.py** by **importing**, **the __name__ variable is set as the name of the python script, i.e. File1**.

- Thus, it can be said that **if __name__ == "__main__"** is the **part** of the program that runs when the script is run from the command line using a command like **Python File1.py**.

Code Example 1

The **myclass.py** file contains the definition of a simple class named **MyClass**, which has an **__init__** method to initialize the object with a name and a **greet** method to print a greeting message.

The **main.py** file imports the **MyClass** class from myclass.py, creates an object named obj with the name "Alice", and calls the greet method on that object.

The if **__name__ == "__main__":** block ensures that **main()** function is only called when main.py is executed directly, not when it's imported as a module into another script.

```python
# File: myclass.py

class MyClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")
```

```python
# File: main.py
from myclass import MyClass

def main():
    # Creating an object of MyClass
    obj = MyClass("Alice")

    # Calling the greet method
    obj.greet()

if __name__ == "__main__":
    main()
```

You can **run main.py**, and it will output:

```
Hello, Alice!
```

Code Example 2

The baseclass.py file contains the definition of a simple base class named BaseClass, which has an __init__ method to initialize an attribute x and a display method to print the value of x.

The derivedclass.py file defines a derived class named DerivedClass, which inherits from BaseClass. It has an __init__ method to initialize both x (from the base class) and y, and it overrides the display method to add functionality specific to the derived class.

The main.py file imports DerivedClass from derivedclass.py, creates an object named obj of type DerivedClass, and calls its display method.

When you run main.py, it will output:

```python
# File: baseclass.py

class BaseClass:
    def __init__(self, x):
        self.x = x

    def display(self):
        print("Base class display method")
        print("Value of x:", self.x)
```

```python
# File: derivedclass.py
from baseclass import BaseClass

class DerivedClass(BaseClass):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def display(self):
        super().display()
        print("Derived class display method")
        print("Value of y:", self.y)
```

```python
# File: main.py
from derivedclass import DerivedClass

def main():
    obj = DerivedClass(10, 20)
    obj.display()

if __name__ == "__main__":
    main()
```

```
Base class display method
Value of x: 10
Derived class display method
Value of y: 20
```

Code Example 3

The engine.py file contains the definition of the Engine class, which represents the engine of a car. It has an __init__ method to initialize the horsepower of the engine and a start method to simulate starting the engine.

The car.py file defines the Car class, which represents a car. It has an __init__ method to initialize the make, model, and horsepower of the car's engine. It uses composition to include an instance of the Engine class. It also has a start method to simulate starting the car, which in turn starts the engine.

The main.py file imports Car from car.py, creates an instance of Car, and calls its start method.

When you run main.py, it will output

```
Toyota Camry with 200 horsepower started
Engine started
```

```python
# File: engine.py

class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def start(self):
        print("Engine started")
```

```python
# File: car.py
from engine import Engine

class Car:
    def __init__(self, make, model, horsepower):
        self.make = make
        self.model = model
        self.engine = Engine(horsepower)

    def start(self):
        print(f"{self.make} {self.model} with {self.engine.horsepower} horsepower started")
        self.engine.start()
```

```python
# File: main.py
from car import Car

def main():
    my_car = Car("Toyota", "Camry", 200)
    my_car.start()

if __name__ == "__main__":
    main()
```

Code Example 4

The mixin.py file defines a mixin class ColorMixin that provides color-related functionality.

The shape.py file defines the main class Shape, which aggregates ColorMixin. It has an __init__ method to initialize the color and shape type of the shape.

The main.py file demonstrates the usage of the Shape class. It creates a red circle, prints its description, changes its color to blue, and prints the updated description.

When you run main.py, it will output:

```
This is a red circle.
This is a blue circle.
Printing color from Shape class using super:
blue
```

```python
# File: mixin.py

class ColorMixin:
    def __init__(self, color):
        self.color = color

    def get_color(self):
        return self.color

    def set_color(self, color):
        self.color = color
```

```python
# File: shape.py
from mixin import ColorMixin

class Shape(ColorMixin):
    def __init__(self, color, shape_type):
        super().__init__(color)
        self.shape_type = shape_type

    def describe(self):
        return f"This is a {self.color} {self.shape_type}."

    def print_color(self):
        print("Printing color from Shape class using super:")
        print(super().get_color())
```

```python
# File: main.py
from shape import Shape

def main():
    # Creating a red circle
    circle = Shape("red", "circle")
    print(circle.describe())

    # Changing the color of the circle
    circle.set_color("blue")
    print(circle.describe())

    # Printing color using the new method
    circle.print_color()

if __name__ == "__main__":
    main()
```

Python File Structuring

Introduction

- Having a good project structure allows better management, leading to better a working environment.
- A well structured project will provide ease of debugging and updating, making it more **scalable**.
- Code will become more **maintainable**.
- Overall code will become more **readable** due to shorter lines of codes.
- Structures also make collaborative work much easier, as people don’t have to work on their tasks easier with **less confusion** and **reduce conflict changes**.

Examples of Project Structures

- Flat or Simple
- Src or Standard
- Module-Based

The Flat or Simple Structure

The “flat layout” refers to organizing a project’s files in a folder or repository, such that the various configuration files and import packages are all in the top-level directory.

```
.
├── README.md
├── noxfile.py
├── pyproject.toml
├── setup.py
├── awesome_package/
│   ├── __init__.py
│   └── module.py
└── tools/
    ├── generate_awesomeness.py
    └── decrease_world_suck.py
```

The src layout

The “src layout” deviates from the flat layout by moving the code that is intended to be importable (i.e. import awesome_package, also known as import packages) into a subdirectory. This subdirectory is typically named src/, hence “src layout”.

```
# the src-layout
.
├── README.md
├── noxfile.py
├── pyproject.toml
├── setup.py
├── src/
│   └── awesome_package/
│       ├── __init__.py
│       └── module.py
└── tools/
    ├── generate_awesomeness.py
    └── decrease_world_suck.py
```