Modules & Packages in Python

Module

Hello, learners. Welcome to this important concept on user-defined module & package creation. I hope, now you have built the confidence to use some of the advanced features of the python programming language.

Upon learning about functions in the previous sessions, it's time to learn about modules to better understand packages. It's a hierarchical structure, each package contains numerous modules and each module contains multiple functions.

A Python module is a .py file with function(s) and/or class(es) that you can reuse in your code. You can also create your own.

A module is a file consisting of Python code. It can define functions, classes, and variables. It also can include runnable code. Any Python file can be referenced as a module. A file containing Python code, for example, test.py , is called a module, and the module name would be tested. Modules are of two types:

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

Let us discuss in detail user-defined modules here.

Python, there are three possible ways to define a module:

A module can be written entirely in Python.

The itertools module is an example of a built-in module that is intrinsically present in the interpreter.

A module can be written in C and loaded dynamically at run-time. About which we will talk later.

In all three cases, the import statement is used to access the code of a module.

Hope that you understand the definition of the module.

Now you will learn the definition of the package.

A Python package is a directory of several modules. A package often holds several modules together, along with an __init__.py file.

Python treats directories containing the __init__ .py file as modules.

The __init__ .py file indicates a folder's contents as being part of a Python package. You can't import files from another directory into a Python project without a __init__ .py file.

Now let us discuss the structure of packages.

A Python package can have several sub-packages in it.

Each package can have several modules.

Each module should start with the __init__ .py file.

Each of the modules should be named with the .py file extension.

The modules and packages have some naming rules.

The following rules should be followed when naming Python modules and packages:

1. All lowercase letters
2. Underscore-separated words
3. No word separators (no hyphens)

Look at this example of package & module structure.

Python operations are the package name that will contain sub-packages. Also, __init__.py will be created which will be an empty file. Along with that, the package has sub-packages or modules of names arithmetic, relational and bitwise will containing multiple functions respectively.

You will understand it better with this demo program of the package using the module.

```
#Create a module using demomodule.py

def addition(a, b):
    c = a+b
    return(c)
```

Now you can call the module with the following set of codes.

```
import demomodule #Import the user-defined module
a=int(input("Enter number 1"))
b=int(input("Enter number 2"))
print("The sum is", demomodule.addition(a,b))  #invoke or call the module
```

The first step is to import the file name in the module, in this case, it is a demo module.

To call or invoke the function you have to write the filename dot function.

demomodule.addition(a,b).
The compiler will prompt user input.
The output will have the sum of the two numbers, in this case, it's 10 and 20 and the sum of the numbers is 30.

This same module can be called by using alias names.
This is mostly used when the filenames are long and they are multiple modules that one has to work with. How do you assign alias names to a module?
Simply by adding the word as a new filename or alias name.
Look at the same program we just executed for calling a function in a module.

Here while importing the user-defined module, ‘i’ can define the alias - demomodule as dm.
So when I call the function I will use the alias name and not the original name to invoke the function.
print("The sum is", dm.addition(a,b))
It will prompt for user input, in this case, it’s 34 and 45, giving a sum of 79 as output.

There are some naming rules to follow while calling a module in a package.
The import statement makes the contents of the module available to the caller. The import statement can have different methods, as seen below.

Let us look at them with examples one by one.
The first is the simplest form which we saw in the previous module.
Import <module name>
Second is when several comma-separated modules may be specified in a single import statement:
import <module_name>[, <module_name> ...]
Individual objects from the module can be imported directly into the caller's symbol table using an alternate form of the import statement:

from <module_name> import <name(s)>

You can also import an entire module under an alternate name:

import <module_name> as <alt_name>

It is also possible to import individual objects but enter them into the local symbol table with alternate names:

from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]

Brace yourself since we will see a full-fledged package with modules and functions in it.

We will start by creating a folder MyPackage consisting of the following files.

1. __init__.py   #Create an empty file
2. hello.py
3. arithmetic.py

We will start by writing the three functions in the file and save them as arithmetic.py
Code in arithmetic.py

```
def sum(x,y):
   return x+y

def average(x,y):
   return (x+y)/2

def power(x,y):
   return x**y
```

Similarly, we will now create a new file and write the code to call hello name and save it as hello.py
Code in hello.py

```
def SayHello(name):
   print("Hello ", name)
   return()
```

Now to call the modules in the package, we will import the modules to the package.

```
from MyPackage import arithmetic #Importing the package and the arithmetic.py file
from MyPackage import hello #Importing the package and the hello.py file
```

Then we will call the functions in the package one by one.

```
print(hello.SayHello("Python Package")) #Invocation or call for SayHello function
print("The Sum is:", arithmetic.sum(10,20)) #Invocation or call for sum function
print("The Average is:", arithmetic.average(10,20)) #Invocation or call for average function
print("The power is:", arithmetic.power(10,2)) #Invocation or call for arithmetic function
```

The output will be as follows.

For the hello string function it will be. Hello Python Package

For the sum function, it will be. The Sum is: 30
For the average function, it will be. The Average is: 15.0
And for the power function, it will be. The power is: 100

Now the same modules can be imported into the package using alias names.
Here we have defined ar as the alias name for the arithmetic module and he for the hello module.
The same will be reflected while calling the functions by referring to alias names.

```
print("The Sum is:", ar.sum(10,20)) #Invocation or call for sum function
print("The Average is:", ar.average(10,20)) #Invocation or call for average function
print("The power is:", ar.power(10,2)) #Invocation or call for arithmetic function
```

Please note that the output will remain the same.

I am so happy and proud to see you complete this module. Now you may call yourself a programmer who can handle complex and multiple codes at the same time. See you in the next module to continue with the topic of packages.

Packages & Itertools

Hello and welcome back to the new learning on packages.

As we had mentioned earlier, we shall talk about itertools as a type module.

It's imperative to understand what itertools are. As you remember it is a type of module.

Now, this module implements several list iterator building blocks.
This module is a memory-efficient, fast tool that can be used alone or in combination to develop iterator algebra operations.

- Itertools supports many operations like map(), count(), repeat(), zip_longest(), groupby(), cycle() etc.
- It consists of three types such as infinite iterators (3 functions), combinatorics iterators (4 functions) using the shortest input sequence (12 functions).
- In total itertools supports 19 different functionalities.
- It is useful for larger data set operations with faster computation.

It's better to learn about these types of operations with examples.

Here you will see how the mapping method is a faster way than looping to process a set of code.

Look at the looping method.

```
# Starting time before looping  method
# Defining lists
import time
L1 = [10, 20, 30]
L2 = [5, 6, 7]
t1 = time.time()

# List multiplication using for loop
print("Result:", end = " ")
for i in range(3):
  print(L1[i] * L2[i], end = " ")

# Ending time after looping
# method
t2 = time.time()
print("\nTime taken by for loop: %.5f" %(t2 - t1))
```

This code is a simple code when you have to multiply the numbers of L1 with the numbers present in L2. Now with the use of for loop using the range function, the code is executed to check for ‘i’ in the range until it ends.
Then the t1 and t2 are functions defined to check the time taken by the loop function to execute the code.

The output has the result of multiplications and the time taken by for loop method is 0.00315
Here look at this example to see how the map function works faster than the loop method.

```
# Python program to demonstrate # iterator module

import  operator
import  time

# Defining lists
L1 = [10, 20, 30]
L2 = [ 5, 6, 7]

# Starting time before map
# function
t1 = time.time()

# Calculating result
a, b, c =  map (operator.mul, L1, L2)

# Ending time after map
# function
t2 = time.time()

# Time taken by map function
print ("Result:" , a, b, c)
print ("Time taken by map function: %.5f"      %(t2 - t1)
```

You may notice that the time taken to execute the list multiplication using the map() function is less than the looping method.
Time taken by map function: 0.00008
Note the multiplication result will remain the same.

Let us look at the count operation in itertool.

```
# Python program to demonstrate
# infinite iterators

import  itertools
```

```
# for in loop
for  i in itertools.count(   5, 5): #Starts with five and increments with 5
  if i == 50:  #If it meets the value 50
    break       #Breaks the infinite iterator count()
  else:
    print (i, end = " " )
```

Here the itertool count operation will start with 5 and increment it with 5 until the value of i is equal to 50. Once the if condition is met, the for loop will break and print the series of numbers as output.
The output is
5 10 15 20 25 30 35 40 45
As you can see how the iterloop count operation fastens the iteration.

Similarly, look at the cycle operation of iterloop.

```
# Python program to demonstrate
# infinite iterators

import  itertools

count =  0

# for in loop
for  i in itertools.cycle(   'HI'): #Repeats the string 'HI' infinitely
  if count >  5:   # Checks for the time times count
    break          #breaks the infinite iterator cycle()
  else:
    print (i, end =  " " )
    count +=  1
```

Now here is this code to demonstrate infinite iterators using cycle operation.
The for loop will repeat the string Hi until the count is less than 5. The for loop will break when the infinite iterator cycle() condition is met and prints Hi.
The Output is
H I H I H I

Now that you have learned about what modules and packages are, it's equally important to know their advantages to apply it.
Python modules and Python packages are two tools for modular programming in Python. Modularizing Python code in a large application has various advantages:

- Simplicity: Rather than focusing on the entire problem, a module usually focuses on one small segment of it.
- Maintainability: Modules are usually built to establish logical boundaries between

different problem domains.

- This makes it easier for a large group of programmers to collaborate on a large application.
- Reusability: A single module functionality can be easily used by other parts of the application.
- Scoping: Modules usually have their namespace, which helps to avoid identifier collisions in different parts of a program.

I hope you are feeling proud of yourself to be able to complete this module with a great understanding of the Python language and its applications.