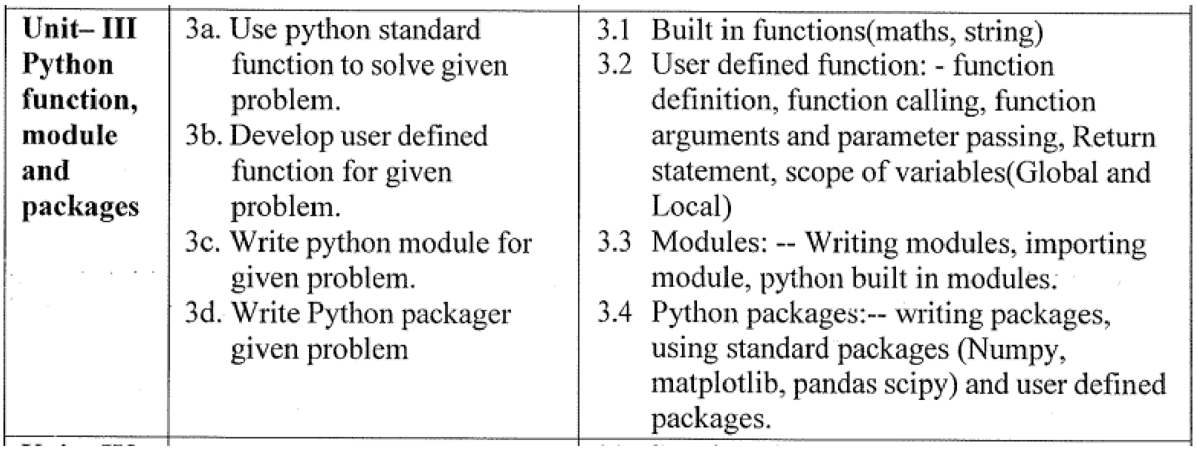
Chapter-3
Python Function, Module and Packages

3.1 Built in functions 2
Math function: 2
Builtin string functions: 4
3.2 Functions: 4
Function Definition: 4
User defined function: 6
Function calling: 7
Function arguments: 8
Positional arguments: 10
Function with Return value: 11
Local and Global variables: 13
Parameter passing/Arguments: 15
Pass by Object Reference 15
3.3 Modules: 16
Write/Create a Module 17
Module attributes: 17
import a module 17
Built in python modules 18
The string Module 20
sring capwords() function 21
The Math Module 21
Use of any four methods of math module: 22
3.4 Packages: 23
Write/Create a package: 23
Standard packages: 27

UNIT-III

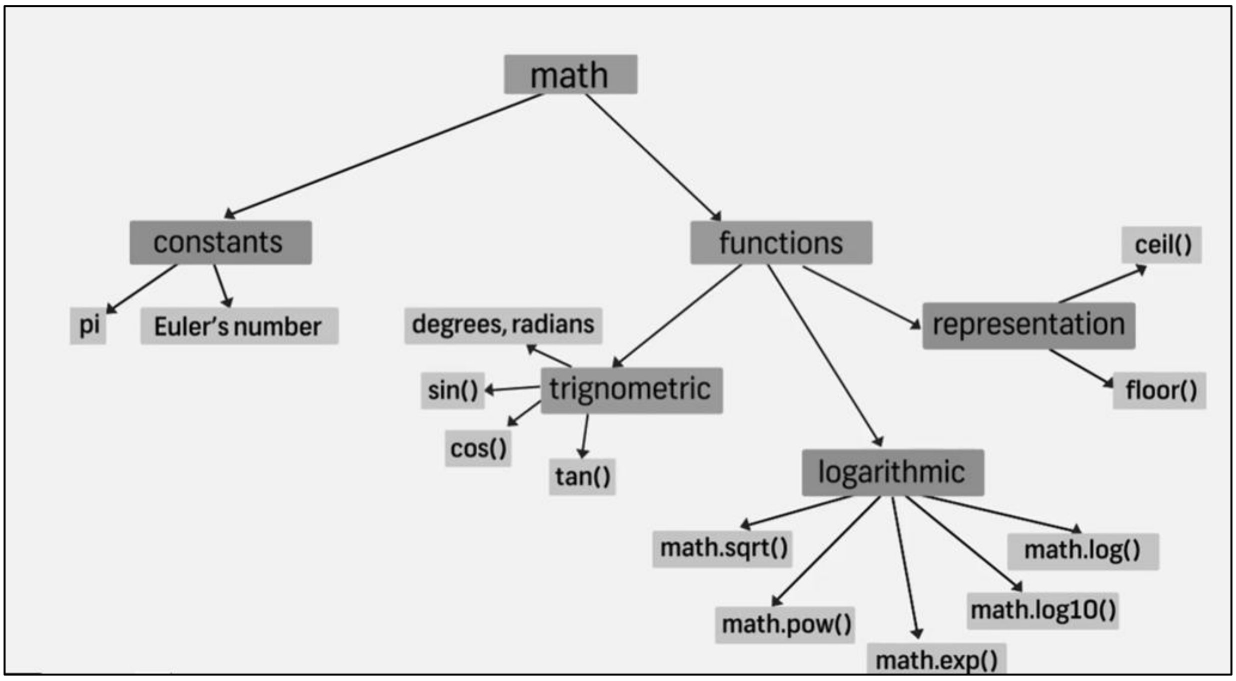
Python Function, Module and Packages

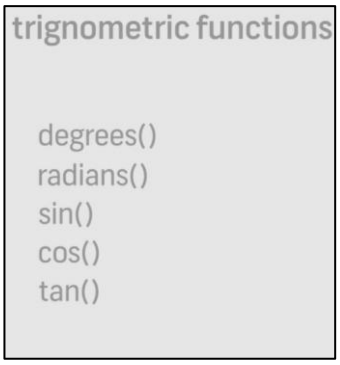


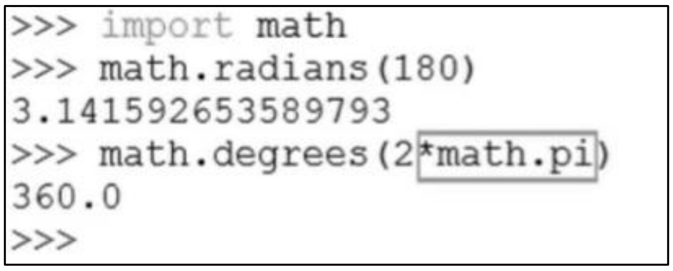
Unit– III Python function, module and packages	3a. Use python standard function to solve given problem. 3b. Develop user defined function for given problem. 3c. Write python module for given problem. 3d. Write Python packager given problem	3.1 Built in functions(maths, string) 3.2 User defined function: - function definition, function calling, function arguments and parameter passing, Return statement, scope of variables(Global and Local) 3.3 Modules: -- Writing modules, importing module, python built in modules. 3.4 Python packages:-- writing packages, using standard packages (Numpy, matplotlib, pandas scipy) and user defined packages.

3.1 <u>Built in functions</u>

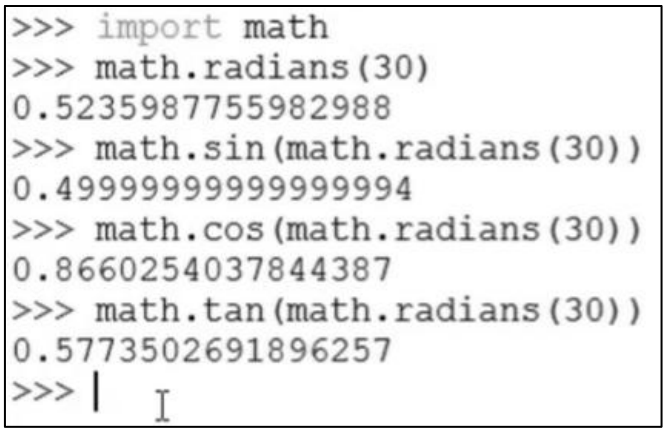
<u>Math function:</u>



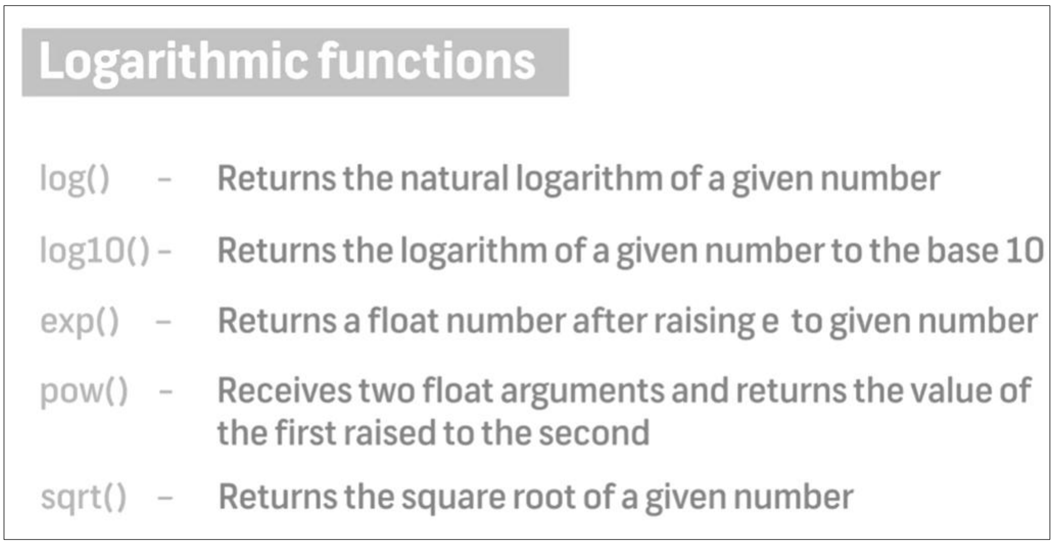

trignometric functions
degrees()
radians()
sin()
cos()
tan()



```
>>> import math
>>> math.radians(180)
3.141592653589793
>>> math.degrees(2*math.pi)
360.0
>>>
```

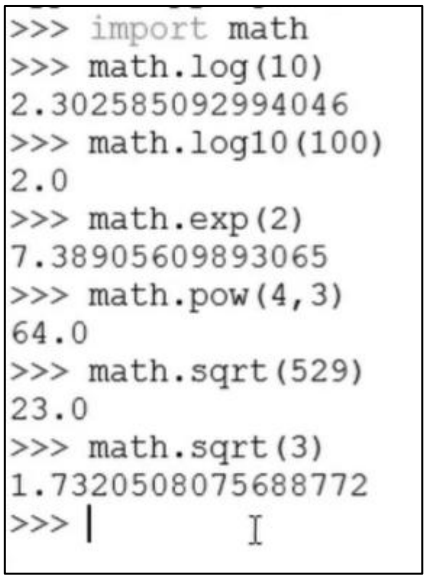


```
>>> import math
>>> math.radians(30)
0.5235987755982988
>>> math.sin(math.radians(30))
0.49999999999999994
>>> math.cos(math.radians(30))
0.8660254037844387
>>> math.tan(math.radians(30))
0.5773502691896257
>>>
```

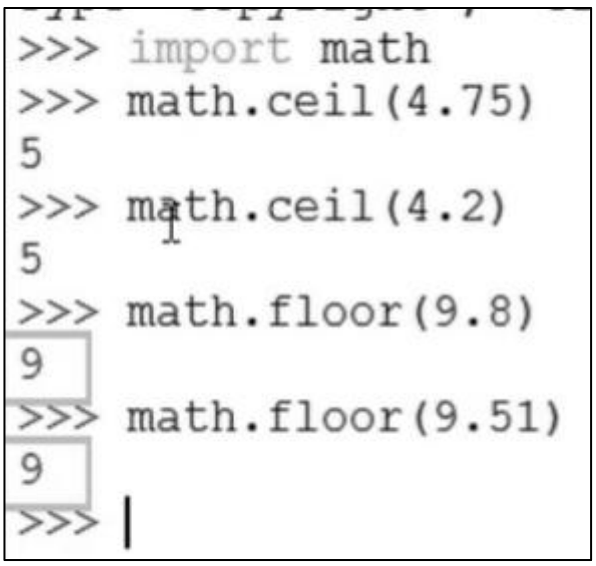


Logarithmic functions

log()	–	Returns the natural logarithm of a given number
log10()	–	Returns the logarithm of a given number to the base 10
exp()	–	Returns a float number after raising e to given number
pow()	–	Receives two float arguments and returns the value of the first raised to the second
sqrt()	–	Returns the square root of a given number



```
>>> import math
>>> math.log(10)
2.302585092994046
>>> math.log10(100)
2.0
>>> math.exp(2)
7.38905609893065
>>> math.pow(4,3)
64.0
>>> math.sqrt(529)
23.0
>>> math.sqrt(3)
1.7320508075688772
>>>
```



```
>>> import math
>>> math.ceil(4.75)
5
>>> math.ceil(4.2)
5
>>> math.floor(9.8)
9
>>> math.floor(9.51)
9
>>>
```

The min() and max() functions can be used to find the lowest or highest value in an iterable:

Example

```
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

Output:

```
5
25
```

The abs() function returns the absolute (positive) value of the specified number:

Example

```
x = abs(-7.25)

print(x)
```

Output:

```
7.25
```

The pow(*x*, *y*) function returns the value of x to the power of y (x^y).

Example

Return the value of 4 to the power of 3 (same as 4 * 4 * 4):

```
x = pow(4, 3)

print(x)
```

Output:

```
64
```

Builtin string functions:

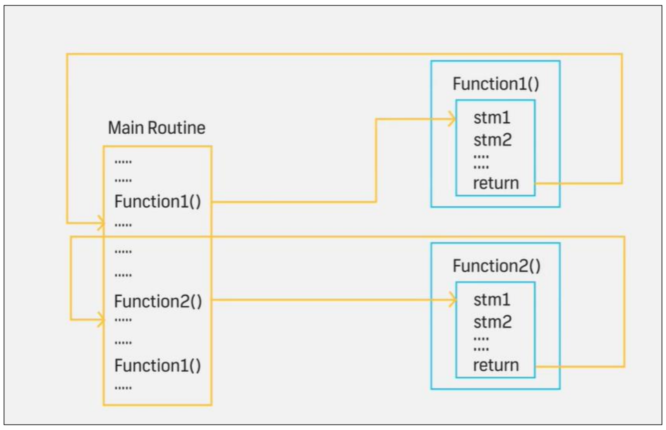
Method	Description
capitalize()	Converts the first character to upper case
count()	Returns the number of times a specified value occurs in a string
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isupper()	Returns True if all characters in the string are upper case
lower()	Converts a string into lower case
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case

3.2 Functions:

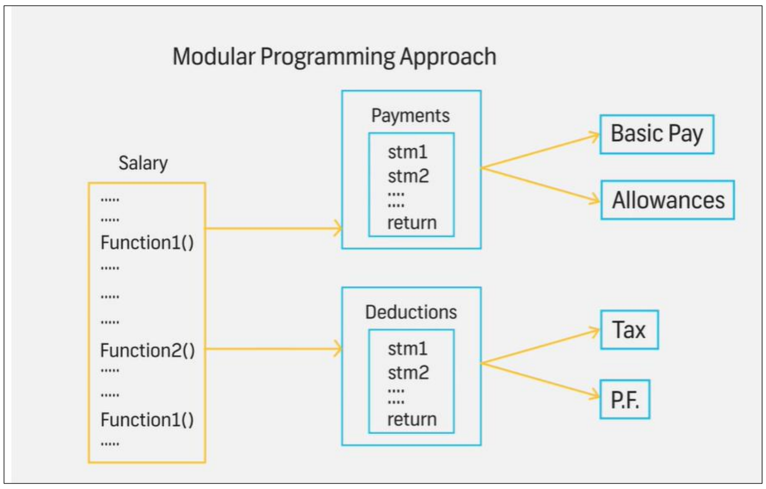
Function Definition:

Functions are **independent and reusable** blocks of instructions. Each function is a separate, complete and reusable software component.

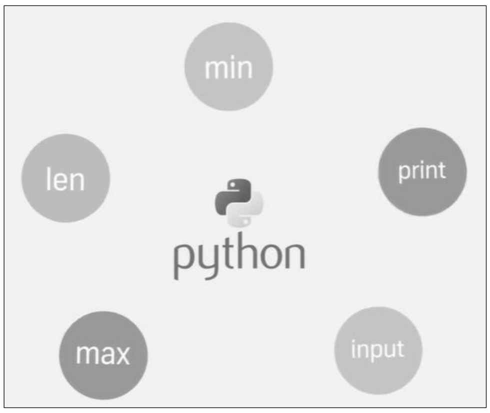
Functions are independent and reusable blocks of instructions. When the programming solution is long and complex, it is good idea to break it into smaller blocks which are independent and reusable. This approach to software development called **modular programming.** It makes the code easy to follow, develop and maintain.



For example: If we were write a program for salary calculation, we would first break it down into two blocks-payments and deductions. Payment block can be further broken down into basic pay and allowances. Deductions into tax and P.F.



Modular programming takes a top-down approach towards software development. Programming solution has main routine through which smaller independent modules or functions are called. Each function is a separate, complete and reusable software component. When we call a function, it performs a specified task and returns a control to the calling routine. So functions can be thought of organization tools that keep your code neat and tidy. Also functions make it easy to reuse the instructions that you have created. Python offers several built in functions, infact we have already used such as



<u>User defined function:</u>

We can create our own functions are known as **user defined functions**.

Syntax of Function without arguments:

```
def function_name() :
   statement_1
   statement_2
   ....
```

Syntax of Function with arguments:

```
def function_name(argument1, argument2, ...) :
   statement_1
   statement_2
   ....
   return expression

function_name(arg1, arg2)
```

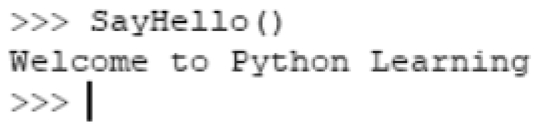
In Python the return statement (the word return followed by an expression.) is used to return a value from a function, return statement without an expression argument returns none.

- We use **<u>def</u>** keyword to define a new function, this is followed by a suitable identifier for the function. The identifier is followed by round brackets. If you want, you can add parameters within these brackets. And finally statement is ended with a colon.
- With a start a new indented block in next line. In this block, the first statement is an explanatory string which describes the functionality. It is called **<u>docs string</u>** and it is optional. It is somewhat similar to comment.
- The statement that follow which perform a certain task form the body of the function. The last statement in the function block has the keyword **<u>return</u>.** It senses the execution back to the calling environment.

```
def SayHello():
    "This will display a greeting message"
    print ("Welcome to Python Learning"
    return
```

Now that we have created a function called SayHello,let us call it(**Function calling**). We simply use identifier with round bracket.

Output:



```
>>> SayHello()
Welcome to Python Learning
>>> |
```

Code Challenge

It's your chance to transform yourself into Harry Potter! Define a function called IPotter which will replace the name Harry with your name in the following string:
Looking Slughorn straight in the eye, **Harry** leant forwards a little.

```
def IPotter():

    "This function morphs anyone into Harry Potter!"

    name="RAVI"

    print("Looking Slughorn straight in the eye, {} leant forwards a
little.".format(name))

    return
```

Output:

```
>>> IPotter()

Looking Slughorn straight in the eye, RAVI leant forwards a little.

>>>
```

<u>Function calling:</u>

- **<u>Create a function in Python</u>:**

  To create a function, we need to use a **def** keyword to declare or write a function in Python. Here is the syntax for creating a function:
  Syntax:
  ```
  def function_name():
  Statement to be executed
  return statement
  ```
  Example:
  Let's create a Myfun.py function program in Python.

  ```
  def myFun(): # define function name

      print(" Welcome to Python Programming")
  ```

```
myFun() # call to print the statement
```

Output:

```
Welcome to Python Programming
```

- **Calling a function:**

  Once a function is created in Python, we can call it by writing **function_name()** itself or another function/ nested function. Following is the syntax for calling a function.

Syntax:

```
Def function_name():
      Statement1
function_name() # directly call the function
```

Example:

```
def MyFun():
   print("Hello World")
   print(" Welcome to Python Programming ")

MyFun() # Call Function to print the message.
```

Output:

```
Hello World
 Welcome to Python Programming
```

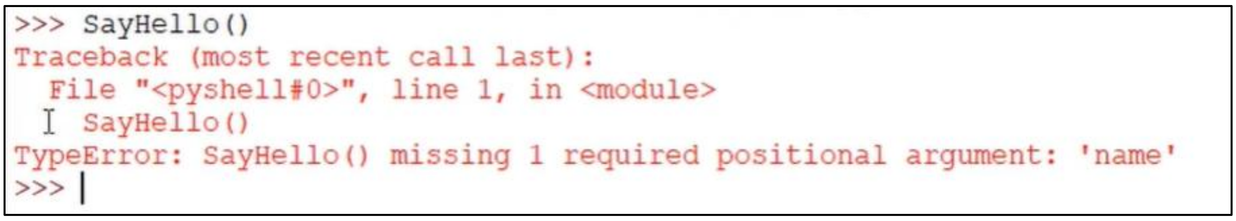
Function arguments:

Functions should also be flexible and allow you to do more than just one thing. Otherwise, you need to create a lot of functions that vary by the data they have used rather than the functionality they provide.

It is possible to define a function to receive data using parameters also called arguments which function can use when performing the task. Using arguments, doesn’t mean that you have to fight with a function. Arguments only help you to create a function that are flexible and use different data.

Take SayHello() function from the previous section. We can modify it to accept an argument called name by adding it in the parenthesis. So SayHello function is not defined to receive a string parameter called name. We modify print statement within the function to display this received argument. Let us call this function and see what happens. We get an error SayHello missing 1 required positional argument name.

```
def  SayHello():
   "This will display a greeting message"
   print("Hello {}.Welcome to Python Learning.\n".format(name))
   return
```



```
>>> SayHello()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    SayHello()
TypeError: SayHello() missing 1 required positional argument: 'name'
>>> 
```

We have two ways for specifying the arguments.

- **Simply include argument in parenthesis while calling the function**

  ```
  def  SayHello(name):
     "This will display a greeting message"
     print("Hello {}.Welcome to Python Learning.\n".format(name))
     return
  ```

  Output:

  ```
  >>> SayHello("Srikar")
  Hello Srikar.Welcome to Python Learning.
  ```

- **The other way is to use variables. When we call the function, It takes the variable as an argument**

  ```
  def  SayHello(name):
     "This will display a greeting message"
     print("Hello{}.Welcome to Python Learning.\n".format(name))
     return
  friend=input("Enter your name:")
  SayHello(friend)
  ```

  Output:

  ```
  Enter your name:Mohit
  Hello Mohit.Welcome to Python Learning.
  ```

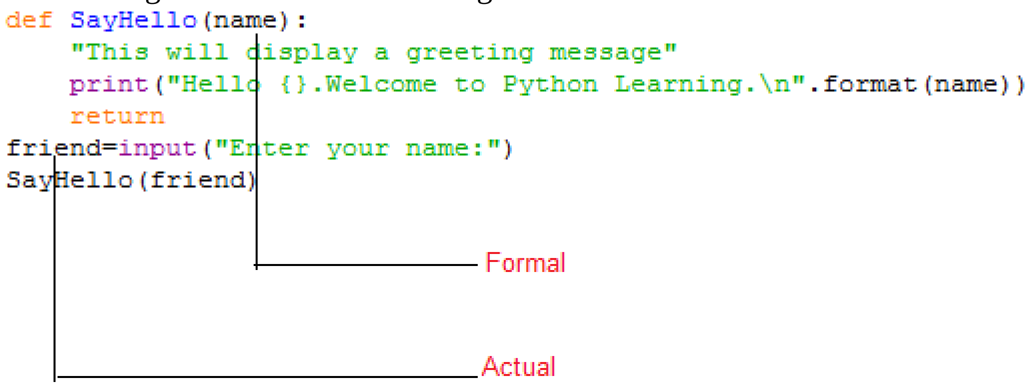
  Output:

  ```
  Enter your name:Lakshmi
  Hello Lakshmi.Welcome to Python Learning.
  ```

- **There are two kinds of arguments.**
  - Formal- Just like plug points
  - Actual- As plugs

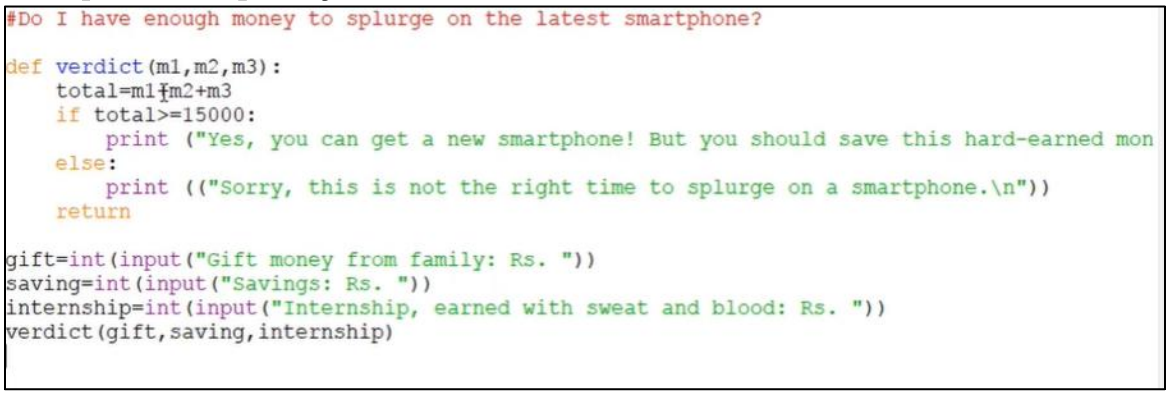
  Formal: The arguments which are used in function definition are formal argments

  Actual- Arguments used when calling the functions

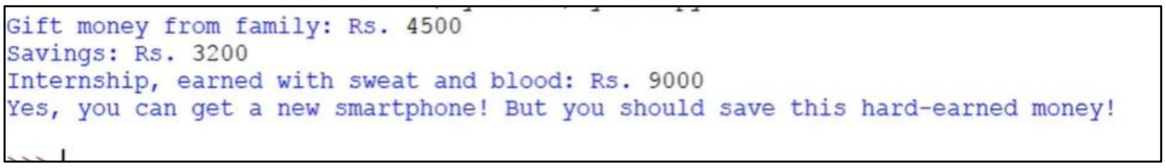


```
def SayHello(name):
    "This will display a greeting message"
    print("Hello {}.Welcome to Python Learning.\n".format(name))
    return
friend=input("Enter your name:")
SayHello(friend)
```

Example of multiple arguments:

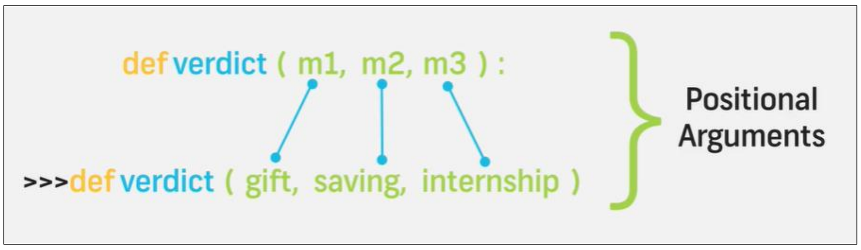


```
#Do I have enough money to splurge on the latest smartphone?

def verdict(m1,m2,m3):
    total=m1+m2+m3
    if total>=15000:
        print ("Yes, you can get a new smartphone! But you should save this hard-earned mon
    else:
        print (("Sorry, this is not the right time to splurge on a smartphone.\n"))
    return

gift=int(input("Gift money from family: Rs. "))
saving=int(input("Savings: Rs. "))
internship=int(input("Internship, earned with sweat and blood: Rs. "))
verdict(gift,saving,internship)
```



```
Gift money from family: Rs. 4500
Savings: Rs. 3200
Internship, earned with sweat and blood: Rs. 9000
Yes, you can get a new smartphone! But you should save this hard-earned money!
```

Positional arguments:


def verdict (m1, m2, m3) :
>>>def verdict (gift, saving, internship)
Positional Arguments

When we call function with arguments, we had to make sure that we provided the same number of actual arguments and also have matching data types.Such arguments are called positional arguments.
If there is a mismatch in either the number or type of argument, we get an error.

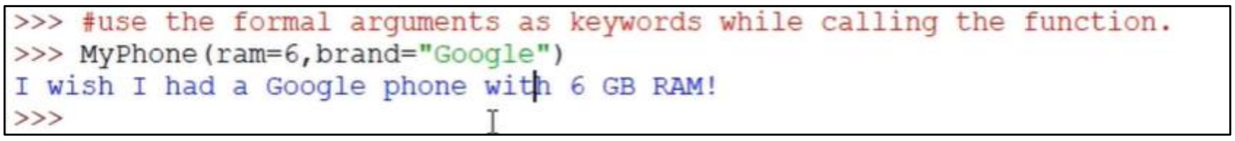
```
def Myphone(brand,RAM):
    print("I wish I had a {:s} phone with {:d} GB RAM".format(brand,RAM))
    return
```

Output:

```
>>> Myphone("iphone",6)
I wish I had a iphone phone with 6 GB RAM
>>> Myphone("iphone")
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    Myphone("iphone")
TypeError: Myphone() missing 1 required positional argument: 'RAM'
>>> Myphone(6,"iphone")
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    Myphone(6,"iphone")
  File "C:/Users/Yadunandan/AppData/Local/Programs/Python/Python38-32/1.py",
line 2, in Myphone
    print("I wish I had a {:s} phone with {:d} GB RAM".format(brand,RAM))
```

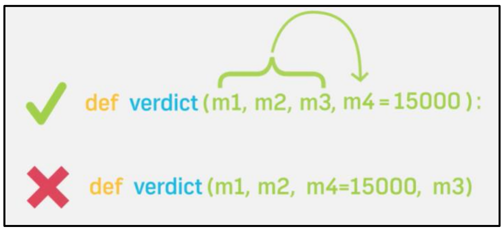
ValueError: Unknown format code 's' for object of type 'int'

We can use formal argument as keywords while calling function.In our example, the formal arguments are brand and ram.Notice that we have switched the position of both arguments, however function works because values are explicitly assigned to arguments.



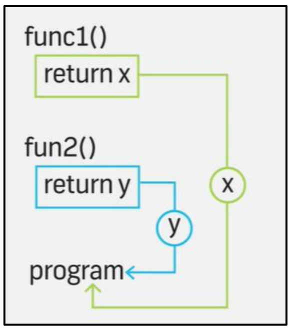
```
>>> #use the formal arguments as keywords while calling the function.
>>> MyPhone(ram=6,brand="Google")
I wish I had a Google phone with 6 GB RAM!
>>>
```

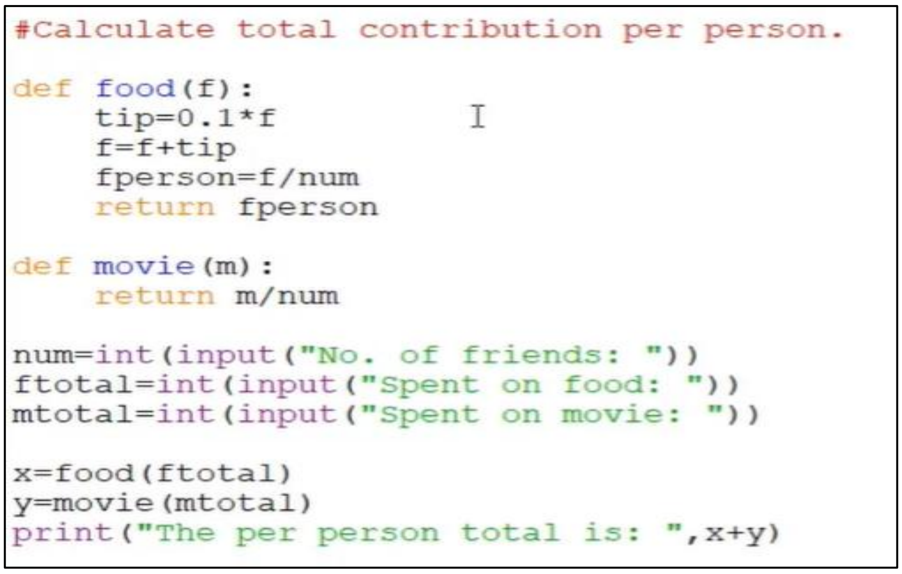
In python, default argument must come always after the required arguments. So in the example we used m4 must be written after m1, m2, m3 because it has default value specified. If this is not followed, the function will throw an error during execution.


✓ def verdict (m1, m2, m3, m4 = 15000):
✗ def verdict (m1, m2, m4=15000, m3)

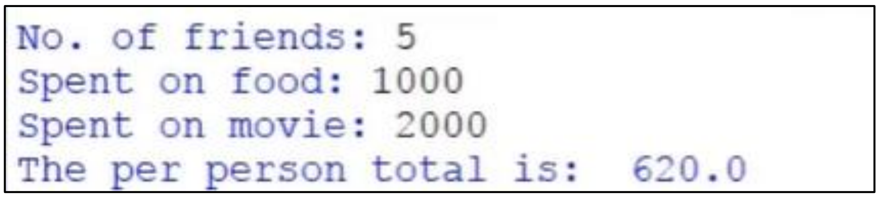
Function with Return value:

We know that return is used to mark the end of the function, indicating that program flow returning to the calling environment. However even if return is not used, the program control goes back to calling environment. So why we use it? We use it to return a pass a value or the result generated by the function back to the calling environment.





```
#Calculate total contribution per person.

def food(f):
    tip=0.1*f
    f=f+tip
    fperson=f/num
    return fperson

def movie(m):
    return m/num

num=int(input("No. of friends: "))
ftotal=int(input("Spent on food: "))
mtotal=int(input("Spent on movie: "))

x=food(ftotal)
y=movie(mtotal)
print("The per person total is: ",x+y)
```



```
No. of friends: 5
Spent on food: 1000
Spent on movie: 2000
The per person total is:  620.0
```

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

Example:

```
def add(a, b):

    # returning sum of a and b
    return a + b
# calling function
res = add(2, 3)
print("Result of add function is {}".format(res))
```

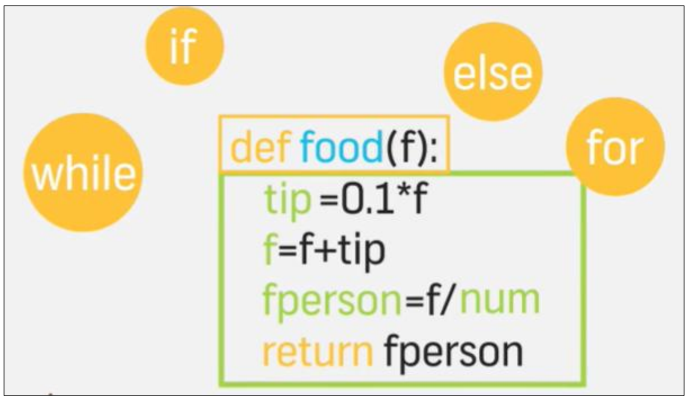
Output:

```
Result of add function is 5
```

Now commenting the return statement and run it.

***Function can be defined to accept any number of arguments but it can return only one value.**

Local and Global variables:



```
def food(f):
    tip =0.1*f
    f=f+tip
    fperson=f/num
    return fperson
```

In general any variable that is defined in the block such as in if, else, while, for or a function is variable only in that block. It is not accessible outside the block. Such a variable is called as **local variable.**

Let us take a simple example:

```
def a():
        tag="AO"
        print(tag)
        return
a()
print(tag)
```

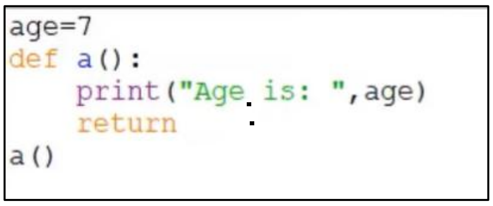
In this example, function a() has a variable called tag. This is a local variable. If we try to access the variable tag outside of a function, we get a name error.Here variable tag is local variable of a function a().

Output:

```
AO
Traceback (most recent call last):
  File "C:/Users/LATA/even,py.py", line 6, in <module>
    print(tag)
NameError: name 'tag' is not defined
```

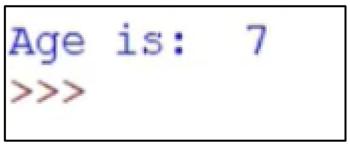
Global variale:

Variables which are outside a any function block are called as **Global variables**. Their values are accessible from any function.



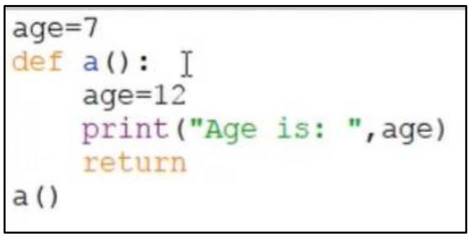
```
age=7
def a():
    print("Age is: ",age)
    return
a()
```

$\longrightarrow$

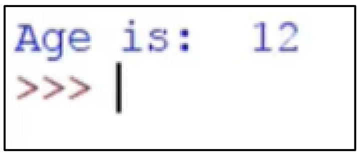


```
Age is:  7
>>>
```

Look at this example. We initialise global variable age before a function a().What will happen if we assign another value to global variable age inside a function a(). A new local variable is created within the function a()

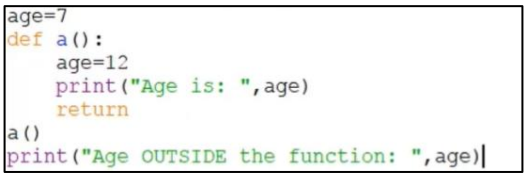


```
age=7
def a():
    age=12
    print("Age is: ",age)
    return
a()
```

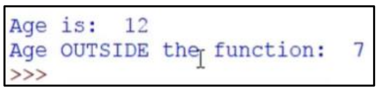


```
Age is:  12
>>> 
```

Global variable remains unchanged. The value of global variable remains same i.e., 7

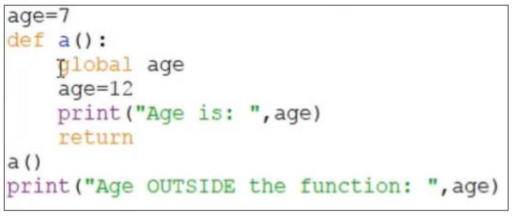


```
age=7
def a():
    age=12
    print("Age is: ",age)
    return
a()
print("Age OUTSIDE the function: ",age)
```

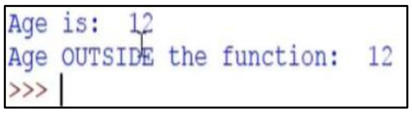


```
Age is:  12
Age OUTSIDE the function:  7
>>>
```

However let us change global variable inside a function. We use keyword global to tell python that we want to use a global variable.

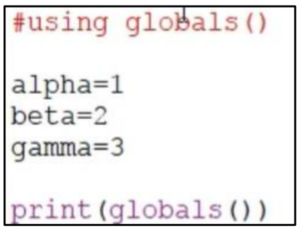


```
age=7
def a():
    global age
    age=12
    print("Age is: ",age)
    return
a()
print("Age OUTSIDE the function: ",age)
```

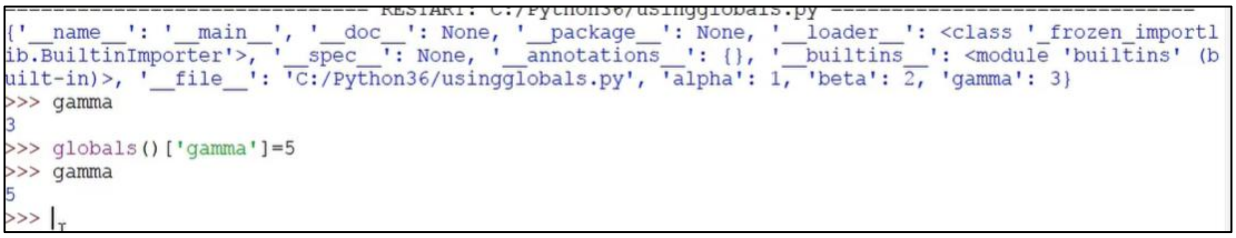


```
Age is:  12
Age OUTSIDE the function:  12
>>> 
```

Using same name for local ang global variable:
This is very much possible with built in function **globals().**
How globals() function works.Let alpha,beta and gamma variables.Lets print the output of globals function.When proram runs, it retuns a dictionary object of all global variables and their respective values.Observe that this includes some default variables in the module as well as global variabls that we have declared.Since this output is dictionary we can access any of the values using a key which is nothing but a variable name.For example we can change a value of gamma using this syntax.

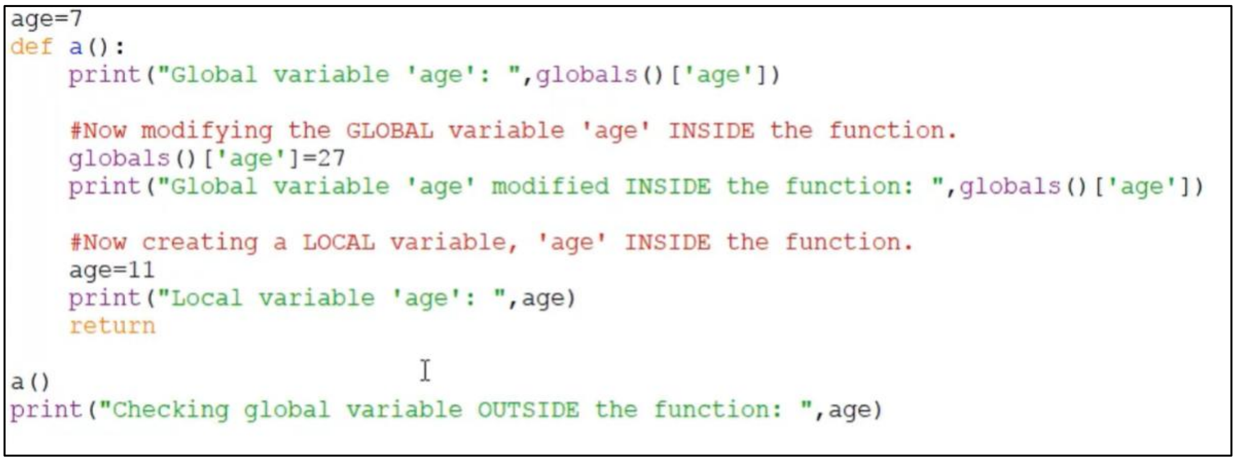


```
#using globals()

alpha=1
beta=2
gamma=3

print(globals())
```



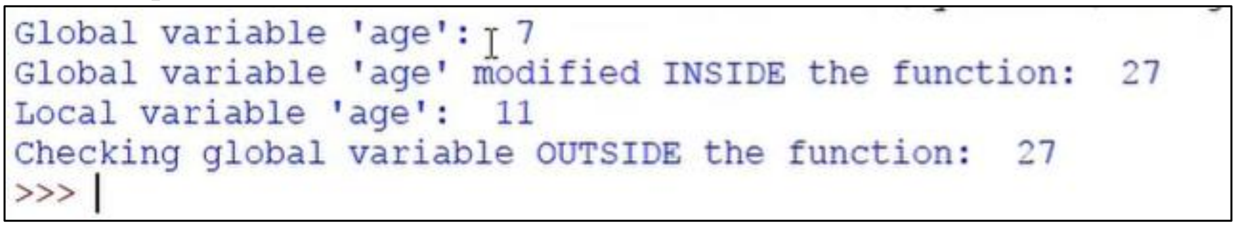
```
=============================== RESTART: C:/Python36/usingglobals.py ===============================
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importl
ib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (b
uilt-in)>, '__file__': 'C:/Python36/usingglobals.py', 'alpha': 1, 'beta': 2, 'gamma': 3}
>>> gamma
3
>>> globals()['gamma']=5
>>> gamma
5
>>> 
```

How to use same name for local and global variable:
Look at this example. We start with global variable age. Then we modify the age inside function globals(). Lets also create a local variable called age. Notice that we wouldn’t prefix with global. So it becomes a global variable.



```
age=7
def a():
    print("Global variable 'age': ",globals()['age'])

    #Now modifying the GLOBAL variable 'age' INSIDE the function.
    globals()['age']=27
    print("Global variable 'age' modified INSIDE the function: ",globals()['age'])

    #Now creating a LOCAL variable, 'age' INSIDE the function.
    age=11
    print("Local variable 'age': ",age)
    return

a()
print("Checking global variable OUTSIDE the function: ",age)
```

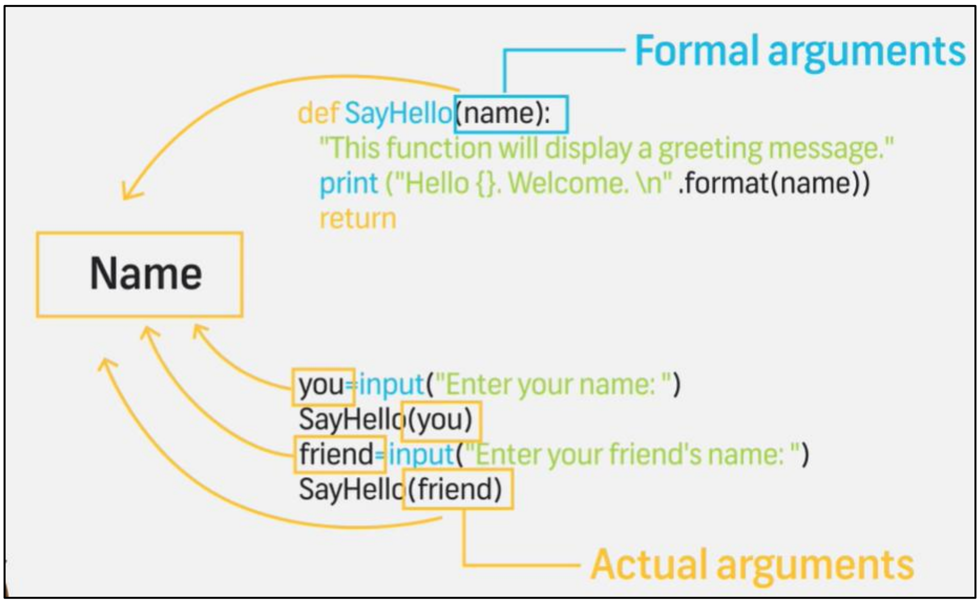
Output:



```
Global variable 'age':  7
Global variable 'age' modified INSIDE the function:  27
Local variable 'age':  11
Checking global variable OUTSIDE the function:  27
>>> |
```

Parameter passing/Arguments:

Arguments in python are passed by reference that means while we distinguish formal and actual arguments depending context they are used in. They basically referred to same variable object. This is because a variable in python essentially a reference to object in a memory.



```
def SayHello(name):
  "This function will display a greeting message."
  print ("Hello {}. Welcome. \n".format(name))
  return

you=input("Enter your name: ")
SayHello(you)
friend=input("Enter your friend's name: ")
SayHello(friend)
```

Pass by Object Reference

A variable in Python is a reference to the object memory. So, both formal and actual arguments refer to the same object. The following code snippet will confirm this.

We have used the id() function earlier. It returns a unique integer corresponding to the identity of an object. In the code displayed below, the id() of a list object before and after passing to a function shows an identical value.

```
def myfunction(newlist):
            print("List accessed in function: ", "id:", id(newlist))
            return

mylist=[10,20,30,40,50]
print("List before passing to function: ", "id:", id(mylist))
```

Output:

```
List before passing to function:  id: 55345960
List accessed in function:  id: 55345960
```

myfunction(mylist)If we modify the list object inside the function and display its contents after the function is completed, changes are reflected outside the function as well.

The following result confirms that arguments are passed by reference to a Python function.

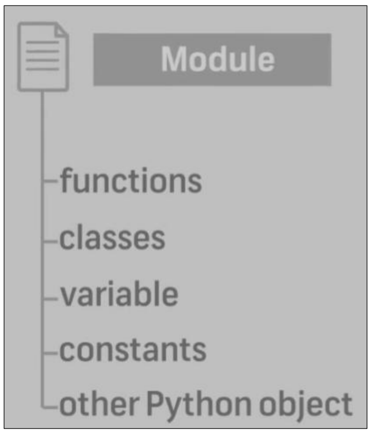
```
def myfunction(newList):
    newList.append(60)
    print("modified list inside function:", newList)
    return

mylist=[10,20,30,40,50]
print("list before passing to function:", mylist)

myfunction(mylist)

print(  print("list after passing to function:", mylist)
```

Output:

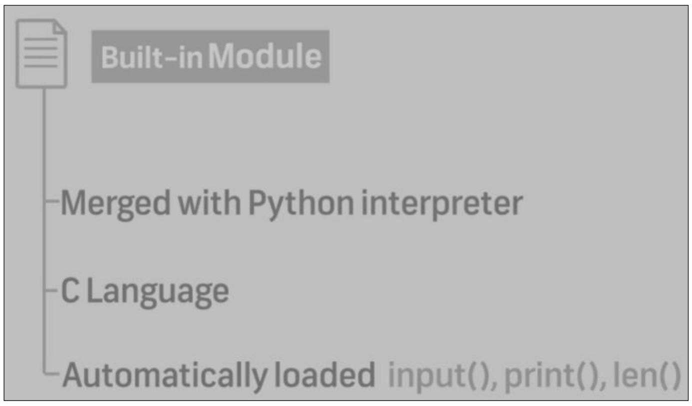
```
list before passing to function: [10, 20, 30, 40, 50]
modified list inside function: [10, 20, 30, 40, 50, 60]
list after passing to function: [10, 20, 30, 40, 50, 60]
```

3.3 Modules:

Python offers a simple solution to organize codes which is called module. A module is a file containing functions, classes, variables, constants and or any other python object.



To use the module, you must tell your program to get the module file and read from it. This process of getting code from another file is called importing. Python provides import statement for this purpose. Simplest way to create a module is to create a file with .py extension. Built in modules are modules merged with python interpreter, they are written in C language, are automatically loaded when the python interpreter starts.


Built-in Module
Merged with Python interpreter
C Language
Automatically loaded input(), print(), len()

Write/Create a Module

To create a module just save the code you want in a file with the file extension .py:
Example
Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module
Now we can use the module we just created, by using the import statement:
Example
Import the module named mymodule, and call the greeting function:

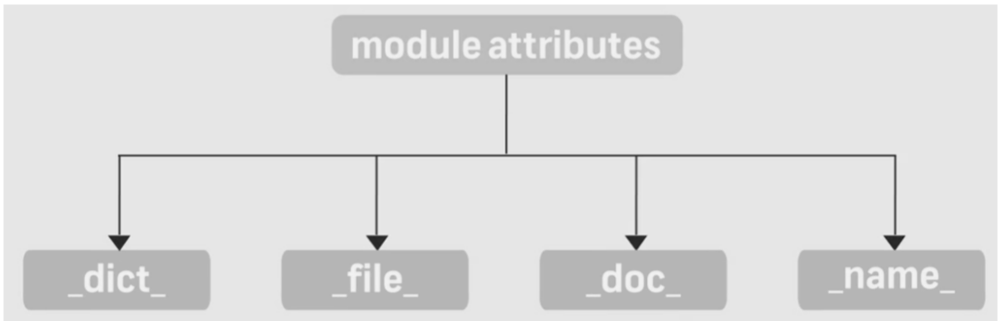
```
import mymodule
mymodule.greeting("Jonathan")
```

Output:

```
Hello, Jonathan
```

Module attributes:

The name attribute tells us the name of module



import a module

Syntax:

When the import is used, it searches for the module initially in the local scope by calling __import__() function. The value returned by the function is then reflected in the output of the initial code.

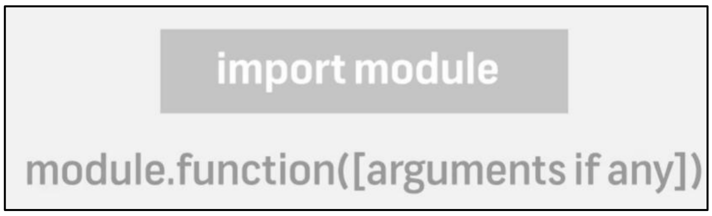
- We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

```
>>> import example
# import statement example
# to import standard module math

import math
print("The value of pi is", math.pi)
```

Output:

```
The value of pi is 3.141592653589793
```

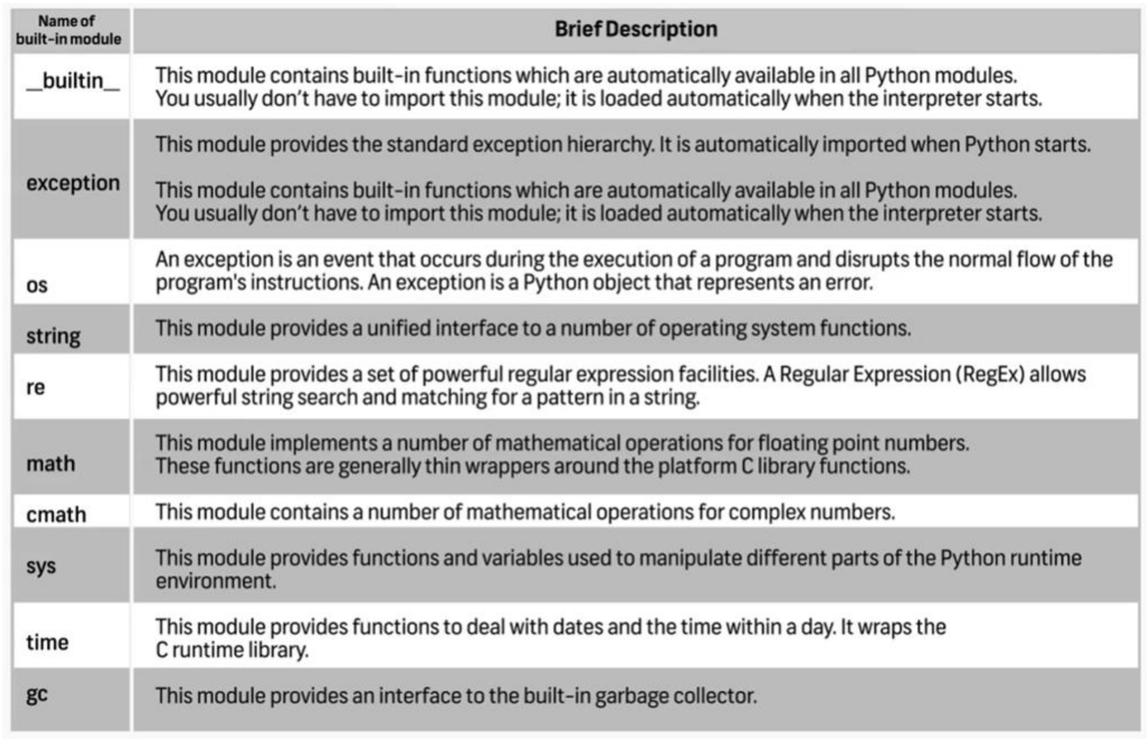
To load resources from another module using import statement. Here is general format of using a function from a module.



```
import module
module.function([arguments if any])
```

Built in python modules

1. os module.
2. random module.
3. math module.
4. time module.
5. sys module.
6. collections module.
7. statistics module.
8. String module



Name of built-in module	Brief Description
__builtin__	This module contains built-in functions which are automatically available in all Python modules. You usually don't have to import this module; it is loaded automatically when the interpreter starts.
exception	This module provides the standard exception hierarchy. It is automatically imported when Python starts. This module contains built-in functions which are automatically available in all Python modules. You usually don't have to import this module; it is loaded automatically when the interpreter starts.
os	An exception is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. An exception is a Python object that represents an error.
string	This module provides a unified interface to a number of operating system functions.
re	This module provides a set of powerful regular expression facilities. A Regular Expression (RegEx) allows powerful string search and matching for a pattern in a string.
math	This module implements a number of mathematical operations for floating point numbers. These functions are generally thin wrappers around the platform C library functions.
cmath	This module contains a number of mathematical operations for complex numbers.
sys	This module provides functions and variables used to manipulate different parts of the Python runtime environment.
time	This module provides functions to deal with dates and the time within a day. It wraps the C runtime library.
gc	This module provides an interface to the built-in garbage collector.

Built in modlules

- builtin
  This module contains built-in functions which are automatically available in all Python modules. You usually don't have to import this module; it is loaded automatically when the interpreter starts.
- **exceptions**
  This module provides the standard exception hierarchy. It is automatically imported when Python starts.
  An exception is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions. An exception is a Python object that represents an error.
- **os**
  This module provides a unified interface to a number of operating system functions.
- **string**
  This module contains a number of functions for string processing.
- **re**
  This module provides a set of powerful regular expression facilities. A Regular Expression (RegEx) allows powerful string search and matching for a pattern in a string.
- **math**
  This module implements a number of mathematical operations for floating point numbers. These functions are generally thin wrappers around the platform C library functions.
- **cmath**

  This module contains a number of mathematical operations for complex numbers.
- **sys**

This module provides functions and variables used to manipulate different parts of the Python runtime environment.

- **time**
  This module provides functions to deal with dates and the time within a day. It wraps the C runtime library.
- **gc**
  This module provides an interface to the built-in garbage collector.

<u>The string Module</u>

It's a built-in module and we have to import it before using any of its constants and classes.

The string module contains a number of functions to process standard Python strings,

```
import string

text = "Monty Python's Flying Circus"

print "upper", "=>", string.upper(text)
print "lower", "=>", string.lower(text)
print "split", "=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text, "Java")
print "count", "=>", string.count(text, "n")
```

Output:

```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying', 'Circus']
join => Monty+Python's+Flying+Circus
replace => Monty Java's Flying Circus
find => 6 -1
count => 3
```

<u>Using the string Module to Convert Strings to Number:</u>

```
import string

print int("4711"),
print string.atoi("4711"),
print string.atoi("11147", 8), # octal
print string.atoi("1267", 16), # hexadecimal
print string.atoi("3mv", 36) # whatever...

print string.atoi("4711", 0),
print string.atoi("04711", 0),
print string.atoi("0x4711", 0)

print float("4711"),
print string.atof("1"),
print string.atof("1.23e5")
```

```
4711 4711 4711 4711 4711
4711 2505 18193
4711.0 1.0 123000.0
```

The constants defined in the string module.

```
import string

# string module constants
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.hexdigits)
print(string.whitespace)  # ' \t\n\r\x0b\x0c'
print(string.punctuation)
```

Output:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF

!"#$%&'()*+,-./:;?@[\]^_`{|}~
```

sring capwords() function

Python string module contains a single utility function – capwords(s, sep=None). This function split the specified string into words using str.split(). Then it capitalizes each word using str.capitalize() function. Finally, it joins the capitalized words using str.join().

If the optional argument sep is not provided or None, then leading and trailing whitespaces are removed and words are separated with single whitespace. If it's provided then the separator is used to split and join the words.

The Math Module

Python has also a built-in module called math, which extends the list of mathematical functions.
To use it, you must import the math module:

```
import math
```

When you have imported the math module, you can start using methods and constants of the module.
The math.sqrt() method for example, returns the square root of a number:

Example

```
import math
```

```
x = math.sqrt(64)

print(x)
```

Output:

```
8.0
```

The math.ceil() method rounds a number upwards to its nearest integer, and the math.floor() method rounds a number downwards to its nearest integer, and returns the result:

Example

```
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

Output:

```
2
1
```

The math.pi constant, returns the value of PI (3.14...):

Example

```
import math

x = math.pi

print(x)
```

Output:

```
3.141592653589793
```

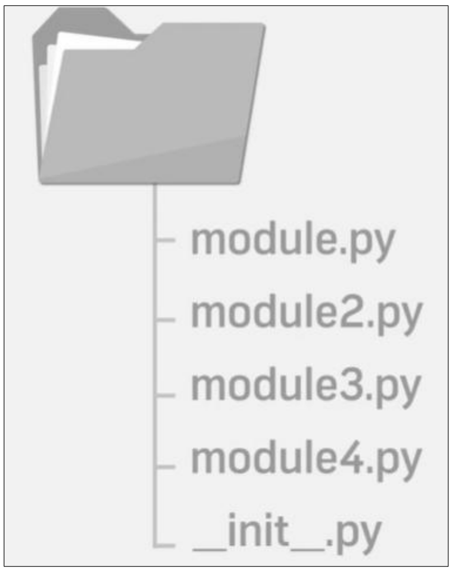
Use of any four methods of math module:

Method	Description
math.fmod()	Returns the remainder of x/y
math.frexp()	Returns the mantissa and the exponent, of a specified number
math.fsum()	Returns the sum of all items in any iterable (tuples, arrays, lists, etc.)
math.gamma()	Returns the gamma function at x
abs(x)	Returns the absolute value of x
sin(x)	Returns the sine of x (x is in radians)

3.4 Packages:

One or more relevant modules can be put into a package. A package is a folder containing one or more modules. Package folder must contain a file called __init__.py. which is the packaging list.__init__.py serves two purposes.

1. Python interpreter recognises folder as a packages if it contains this file.
2. It exposes specified resources from its module to be imported.
   A valid python package can be installed using setupscrits. The script calls setup function from setuptools module.

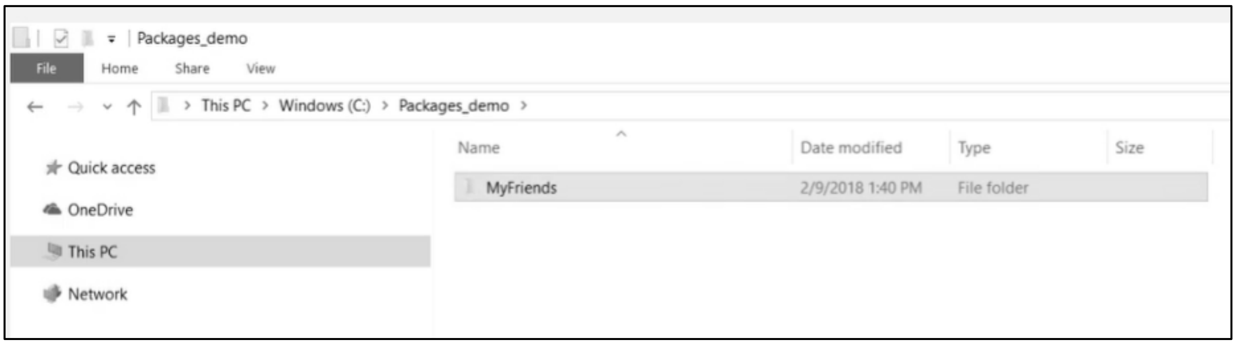


Write/Create a package:

To create a package in Python, we need to follow these three simple steps:

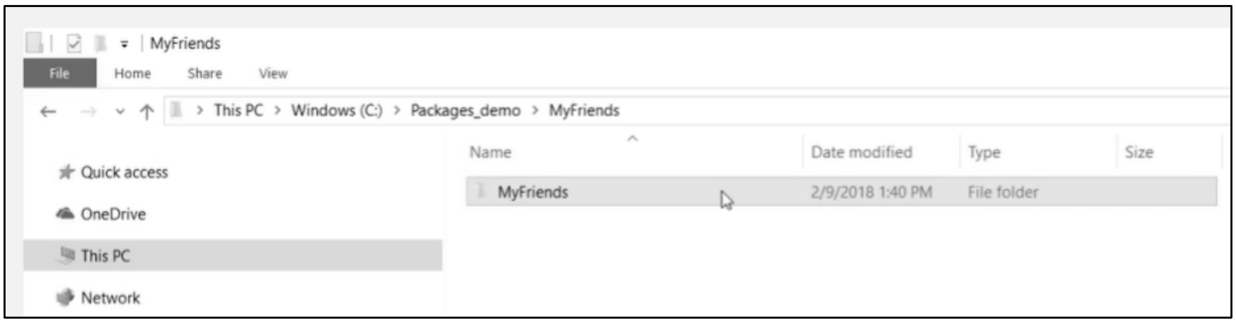
1. First, we create a directory and give it a package name, preferably related to its operation.
2. Then we put the classes and the required functions in it.
3. Finally we create an __init__.py file inside the directory, to let Python know that the directory is a package.

Let us see how to create a package called Myfriends. First we create a folder called Myfriends



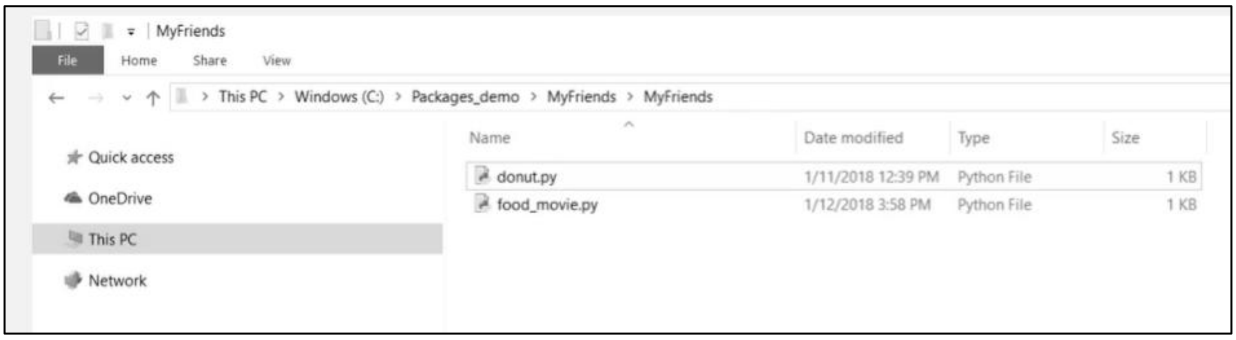
Name	Date modified	Type	Size
MyFriends	2/9/2018 1:40 PM	File folder	

then we create a sub folder with the same name Myfriends.



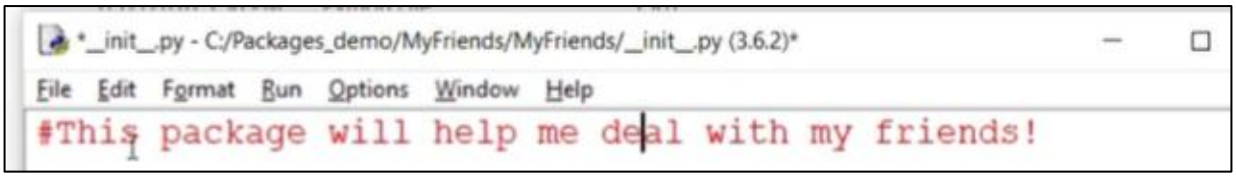
Name	Date modified	Type	Size
MyFriends	2/9/2018 1:40 PM	File folder	

We now placed different modules inside the sub folder.



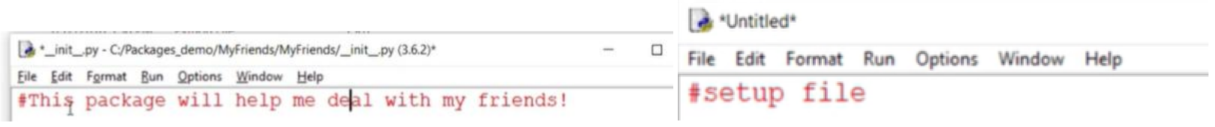
Name	Date modified	Type	Size
donut.py	1/11/2018 12:39 PM	Python File	1 KB
food_movie.py	1/12/2018 3:58 PM	Python File	1 KB

Now let's create the __init__.py file simply open IDLE in the scripting mode and save an empty file with the file name double underscore init double underscore .py. You can also add some comments or define functions from individual module to be made available conveniently.


__init__.py - C:/Packages_demo/MyFriends/MyFriends/__init__.py (3.6.2)
File Edit Format Run Options Window Help

```
#This package will help me deal with my friends!
```

Finally we need to create the set up file which will allow us to install the package.


__init__.py - C:/Packages_demo/MyFriends/MyFriends/__init__.py (3.6.2)
File Edit Format Run Options Window Help
Untitled
File Edit Format Run Options Window Help

```
#This package will help me deal with my friends!
```

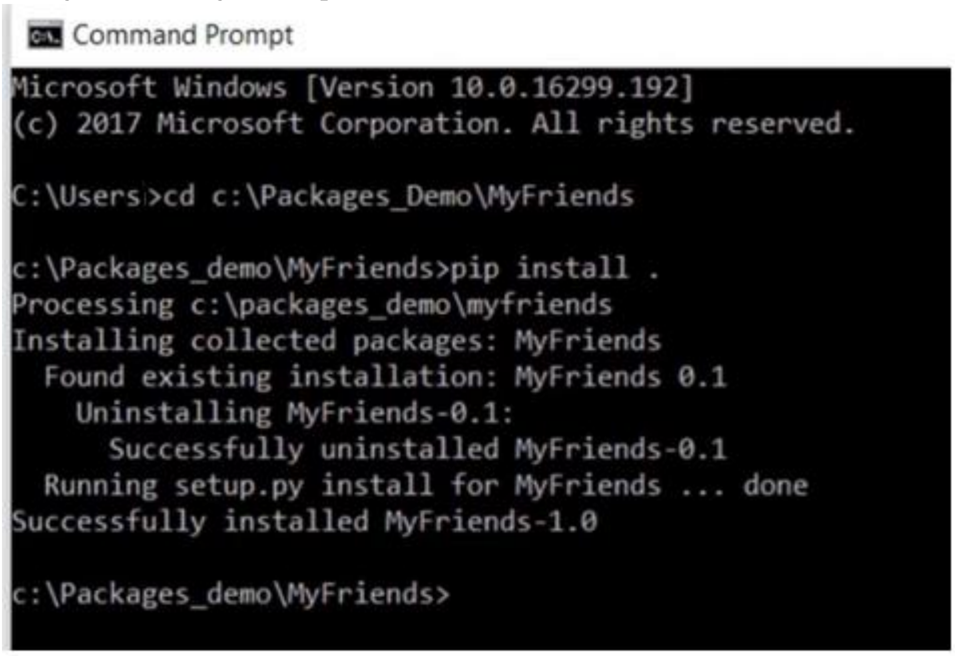
```
#setup file
```

Note that this file must be placed in the parent folder. Basically this script calls the setup function from setuptools module. The setup function takes several argument such as name, version, author etc. The argument zip safe determines whether the package is installed in compressed mode or regular mode.

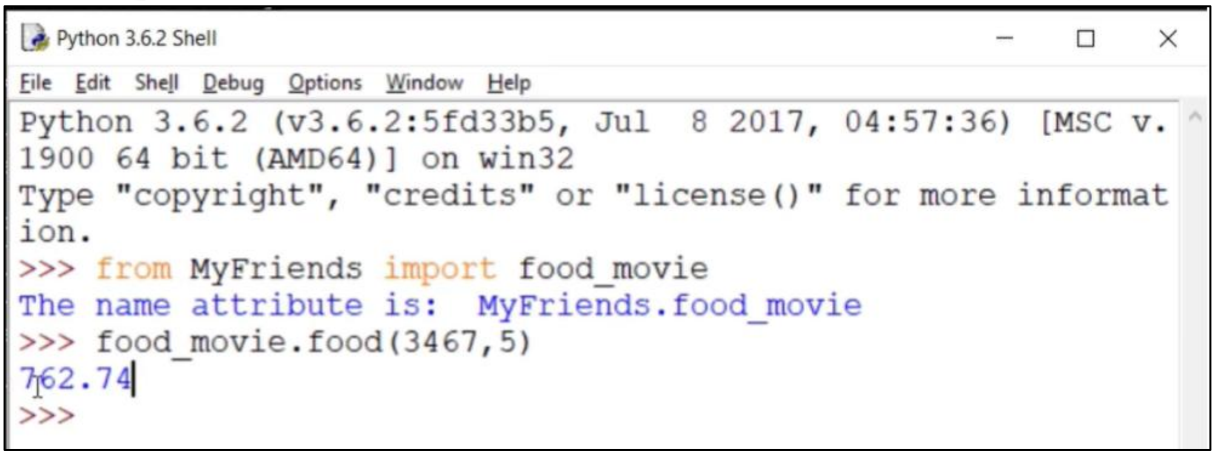
- **Creating a Package Setup File**

```
from setuptools import setup

setup(name='MyFriends', version='1.0', description='A package to calculate expenses when
you hang-out with friends.',
url='#',
    author='V.E.S.P',
author_email='V.E.S.P@VES.AC.IN',
    license='MIT',
    packages=['MyFriends'],
zip_safe=False)
```

Now let's test if we can install the Myfriends package for this we first launch the command prompt then we change to the parent directory Myfriends.We execute command **pip install.** To install the Myfriends package using the utility Myfriends package is now available for use in the system and can be imported from any script. We can try using some of the functions from it to verify this. let say we import the food function It works.



```
Command Prompt
Microsoft Windows [Version 10.0.16299.192]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users>cd c:\Packages_Demo\MyFriends

c:\Packages_demo\MyFriends>pip install .
Processing c:\packages_demo\myfriends
Installing collected packages: MyFriends
  Found existing installation: MyFriends 0.1
    Uninstalling MyFriends-0.1:
      Successfully uninstalled MyFriends-0.1
  Running setup.py install for MyFriends ... done
Successfully installed MyFriends-1.0

c:\Packages_demo\MyFriends>
```



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.
1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more informat
ion.
>>> from MyFriends import food_movie
The name attribute is:  MyFriends.food_movie
>>> food_movie.food(3467,5)
762.74
>>>
```

ONE MORE EXAMPLE:

Let's create a package named Cars and build three modules in it namely, Bmw, Audi and Nissan.

1. **First we create a directory and name it Cars.**
2. **Then we need to create modules**. To do this we need to create a file with the name Bmw.py and create its content by putting this code into it.

```
# Python code to illustrate the Modules
class Bmw:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['i8', 'x1', 'x5', 'x6']

    # A normal print function
    def outModels(self):
        print('These are the available models for BMW')
        for model in self.models:
            print('\t%s ' % model)
```

3. Then we create another file with the name Audi.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Audi:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['q7', 'a6', 'a8', 'a3']

    # A normal print function
    def outModels(self):
        print('These are the available models for Audi')
        for model in self.models:
            print('\t%s ' % model)
```

4. Then we create another file with the name Nissan.py and add the similar type of code to it with different members.

```
# Python code to illustrate the Module
class Nissan:
    # First we create a constructor for this class
    # and add members to it, here models
    def __init__(self):
        self.models = ['altima', '370z', 'cube', 'rogue']

    # A normal print function
    def outModels(self):
        print('These are the available models for Nissan')
        for model in self.models:
            print('\t%s ' % model)
```

5. **Finally we create the __init__.py file.** This file will be placed inside Cars directory and can be left blank or we can put this initialisation code into it.

```
from Bmw import Bmw
from Audi import Audi
from Nissan import Nissan
```

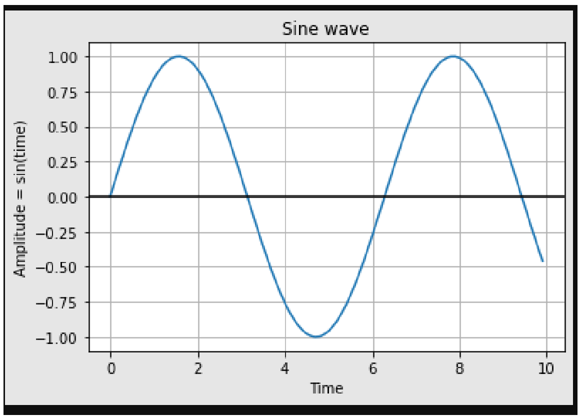
6. Now, let's use the package that we created. To do this make a sample.py file in the same directory where Cars package is located and add the following code to it:

```
# Import classes from your brand new package
from Cars import Bmw
from Cars import Audi
from Cars import Nissan

# Create an object of Bmw class & call its method
ModBMW = Bmw()
ModBMW.outModels()

# Create an object of Audi class & call its method
ModAudi = Audi()
ModAudi.outModels()

# Create an object of Nissan class & call its method
ModNissan = Nissan()
ModNissan.outModels()
```

Standard packages:

1. **Matplotlib:** This library is responsible for plotting numerical data. And that's why it is used in data analysis. It is also an open-source library and plots high-defined figures like pie charts, histograms, scatterplots, graphs, etc.
   plot(x, y)

```
import numpy as np
import matplotlib.pyplot as plot
# Get x values of the sine wave
time       = np.arange(0, 10, 0.1);
# Amplitude of the sine wave is sine of a variable like time
amplitude   = np.sin(time)
# Plot a sine wave using time and amplitude obtained for the sine wave
plot.plot(time, amplitude)
# Give a title for the sine wave plot
plot.title('Sine wave')
# Give x axis label for the sine wave plot
plot.xlabel('Time')
# Give y axis label for the sine wave plot
plot.ylabel('Amplitude = sin(time)')
plot.grid(True, which='both')
plot.axhline(y=0, color='k')
plot.show()
# Display the sine wave
plot.show()
```



2. **Pandas:** Pandas are an important library for data scientists. It is an open-source machine learning library that provides flexible high-level data structures and a variety of analysis tools. It eases data analysis, data manipulation, and cleaning of data. Pandas support operations like Sorting, Re-indexing, Iteration, Concatenation, Conversion of data, Visualizations, Aggregations, etc.
3. **Numpy:** The name "Numpy" stands for "Numerical Python". It is the commonly used library. It is a popular machine learning library that supports large matrices and multi-dimensional data. It consists of in-built mathematical functions for easy computations. Even libraries like TensorFlow use Numpy internally to perform several operations on tensors. Array Interface is one of the key features of this library.
4. **SciPy:** The name "SciPy" stands for "Scientific Python". It is an open-source library used for high-level scientific computations. This library is built over an extension of Numpy. It works with Numpy to handle complex computations. While Numpy allows sorting and indexing of array data, the numerical data code is stored in SciPy. It is also widely used by application developers and engineers.