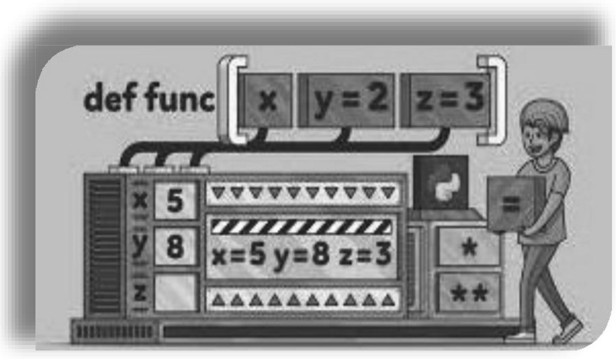
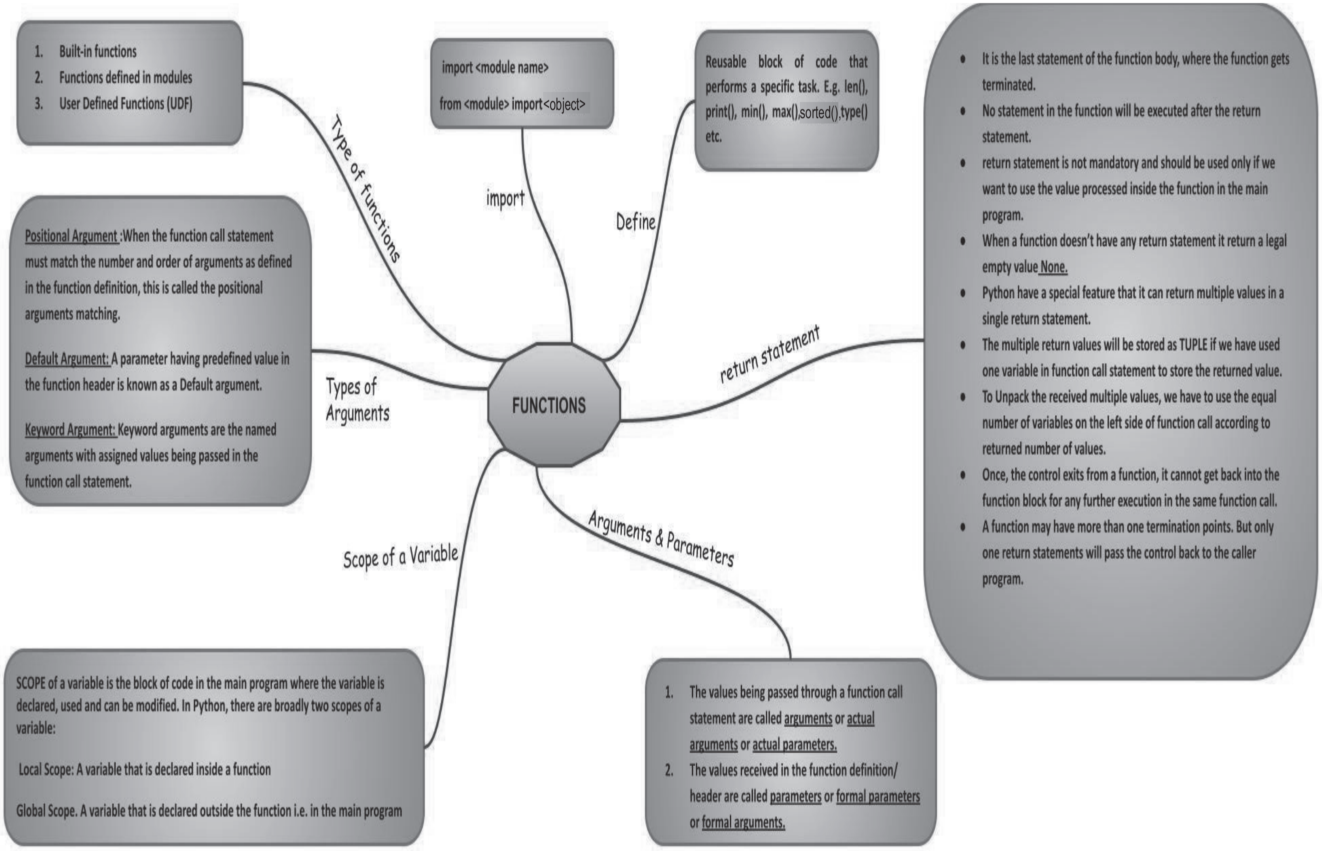
Functions



Topics to be covered

- Introduction
- Types of Functions
- Create User User-defined Function
- Arguments And Parameters
- Default Parameters
- Positional Parameters
- Function Returning Values
- Flow of Execution
- Scope of Variable



1. Built-in functions
2. Functions defined in modules
3. User Defined Functions (UDF)

import <module name>

from <module> import<object>

Reusable block of code that performs a specific task. E.g. len(), print(), min(), max(),sorted(),type() etc.

- It is the last statement of the function body, where the function gets terminated.
- No statement in the function will be executed after the return statement.
- return statement is not mandatory and should be used only if we want to use the value processed inside the function in the main program.
- When a function doesn't have any return statement it return a legal empty value None.
- Python have a special feature that it can return multiple values in a single return statement.
- The multiple return values will be stored as TUPLE if we have used one variable in function call statement to store the returned value.
- To Unpack the received multiple values, we have to use the equal number of variables on the left side of function call according to returned number of values.
- Once, the control exits from a function, it cannot get back into the function block for any further execution in the same function call.
- A function may have more than one termination points. But only one return statements will pass the control back to the caller program.

Positional Argument :When the function call statement must match the number and order of arguments as defined in the function definition, this is called the positional arguments matching.

Default Argument: A parameter having predefined value in the function header is known as a Default argument.

Keyword Argument: Keyword arguments are the named arguments with assigned values being passed in the function call statement.

SCOPE of a variable is the block of code in the main program where the variable is declared, used and can be modified. In Python, there are broadly two scopes of a variable:

Local Scope: A variable that is declared inside a function

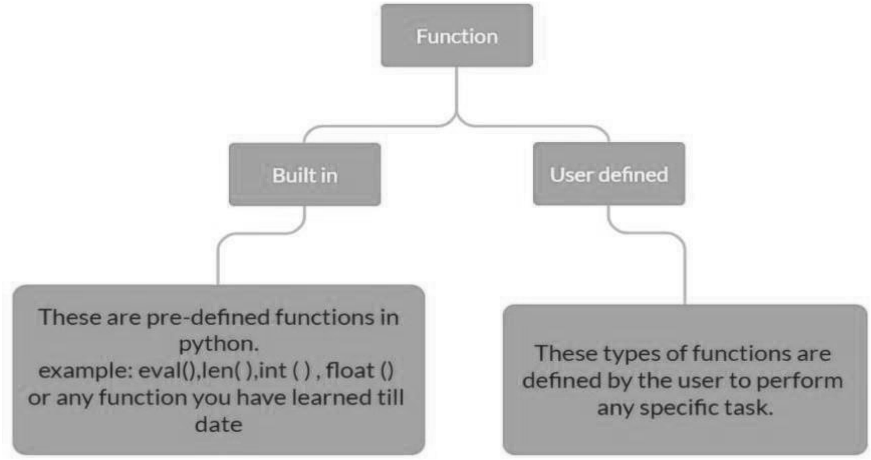
Global Scope. A variable that is declared outside the function i.e. in the main program

1. The values being passed through a function call statement are called arguments or actual arguments or actual parameters.
2. The values received in the function definition/ header are called parameters or formal parameters or formal arguments.

Function

Reusable block of code that performs a specific task. For example: len(), print(), min(), max(), sorted () , type() etc.

Types of Functions


Function
Built in
User defined
These are pre-defined functions in python.
example: eval(),len( ),int ( ) , float ()
or any function you have learned till date
These types of functions are defined by the user to perform any specific task.

Defining a function in Python:

Name the function and specify what to do when the function is called. Python interpreter ignores the function definition until the function is called.

Calling a function:

Calling the function performs the specified actions with the indicated parameters

Function Definition in Python

In Python, a function is defined using the def keyword

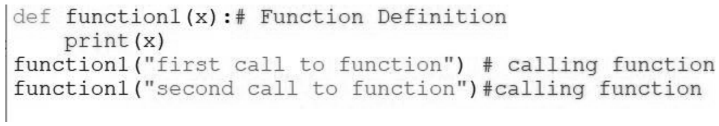
```
def my_function():
  print("Hello from a function")

my_function()
```

- Arguments: Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.
- Actual Parameters (Arguments) are values supplied to the function when it is invoked/called

- Formal Parameters are variables specified in function definitions to receive values from arguments passed during function calls.

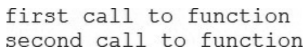
Example 1:

Observe the following code:



```
def function1(x):# Function Definition
    print(x)
function1("first call to function") # calling function
function1("second call to function")#calling function
```

Output:



```
first call to function
second call to function
```

In the above example, a user-defined function “function1” has been defined that receives one argument. Once the function is defined, it can be called any number of times with different arguments.

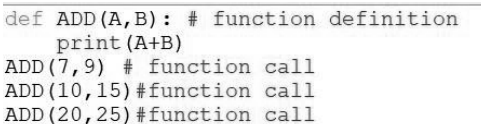
Formal argument: x

Actual argument:

“first call to function “ passed in the first call

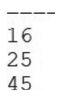
“second call to function” passed in the second call

Example 2: Write a function ADD(A, B) that receives two integer arguments and prints their sum.



```
def ADD(A,B): # function definition
    print(A+B)
ADD(7,9) # function call
ADD(10,15)#function call
ADD(20,25)#function call
```

Output:

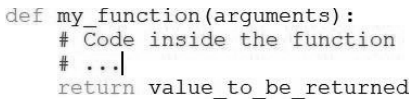


```
----
16
25
45
```

return keyword:

In Python, the `return` keyword is used in functions to specify the value that the function will return when it is called. When a function is executed, it may perform some computations or operations, and the result can be sent back to the caller using the `return` statement.

The basic syntax for using the `return` statement is as follows:

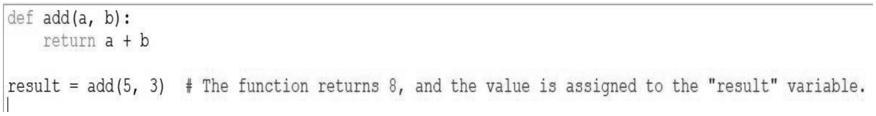


```
def my_function(arguments):
    # Code inside the function
    # ...
    return value_to_be_returned
```

Here's what you need to know about the `return` statement:

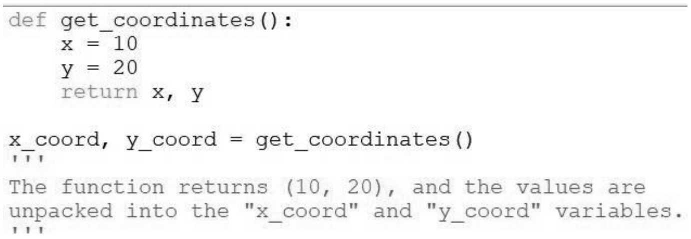
1. **Returning a Value:**
   When you want to return a specific value from the function, you can use the `return` statement followed by the value you want to return.

   Note: The function will stop executing immediately after the `return` statement is encountered, and the value will be passed back to the caller.



```
def add(a, b):
    return a + b

result = add(5, 3)  # The function returns 8, and the value is assigned to the "result" variable.
```

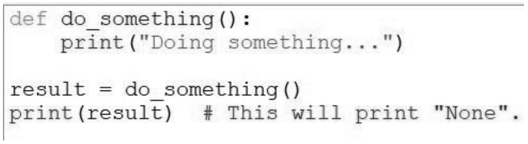
2. **Returning Multiple Values:**
   Python allows you to return multiple values from a function as a tuple. You can simply separate the values with commas after the `return` statement.



```
def get_coordinates():
    x = 10
    y = 20
    return x, y

x_coord, y_coord = get_coordinates()
'''
The function returns (10, 20), and the values are
unpacked into the "x_coord" and "y_coord" variables.
'''
```

3. **Returning None:**
   If a function doesn't have a `return` statement or has a `return` statement without any value, it implicitly returns `None`.

   `None` is a special constant in Python that represents the absence of a value.



```
def do_something():
    print("Doing something...")

result = do_something()
print(result)  # This will print "None".
```

4. **Early Exit with Return:**
   You can use the `return` statement to exit a function early if certain conditions are met. This is useful when you want to terminate the function before reaching the end.

```
def divide(a, b):
    if b == 0:
        return "Cannot divide by zero!"
    return a / b

result1 = divide(10, 2)  # Returns 5.0
result2 = divide(10, 0)  # Returns "Cannot divide by zero!"
```

The `return` statement is a powerful tool that enables functions to produce results and pass data back to the calling code. Understanding how to use it correctly will help you design and implement effective functions in Python.

Scope of a variable:

In Python, the scope of a variable refers to the region of the program where the variable is accessible. The scope determines where a variable is created, modified, and used.

Global Scope:

- Variables defined outside of any function or block have a global scope.
- They are accessible from anywhere in the code, including inside functions.

```
x = 10   # Global variable

def func():
    print(x)  # Accessing the global variable inside the function

def func1():
    print(x)  #Accessing the global variable inside the function

func()    # Output: 10
func1()   # Output:10
```

Local Scope:

- Variables defined inside a function have a local scope.
- They are accessible only within the function where they are defined.
- Local variables are created when the function is called and destroyed when the function returns.

```
def func():
    y = 5   # Local variable
    print(y)

func()    # Output: 5

# y is not accessible outside the function
# print(y) will result in an error
```

Points to be noted:

- When the local variable and global variable have different names: the global variable can be accessed inside the function

```
a=10 #global variable
def fun():
    x=20 #local variable
    y=x+a # global copy of variable a is accessed
    return y
print(fun()) # 30
```

- When local and global variables have the same name: priority is given to the local copy of the variable

```
a=10 #global variable
def fun():
    a=20 #local variable
    y=20+a # local copy of variable a is accessed
    return y
print(fun()) # 40
```

global keyword

In Python, the ***global*** keyword is used to indicate that a variable declared inside a function should be treated as a global variable, rather than a local variable. When you assign a value to a variable inside a function, Python, by default, creates a local variable within that function's scope. However, if you need to modify a global variable within a function, you must use the global keyword to specify that you want to work with the global variable instead.

Here's the basic syntax for using the global keyword:

```
global variable_name
```

For example:

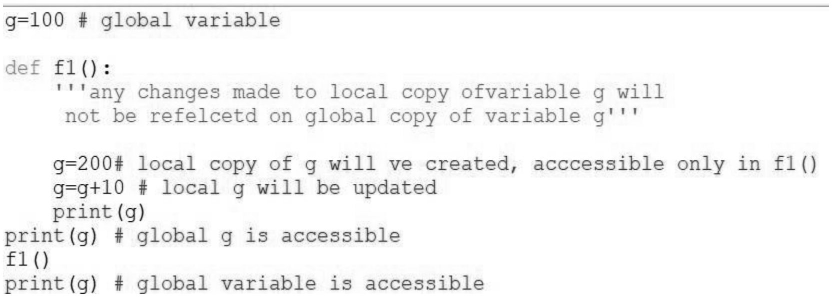
```
global_var = 10

def modify_global():
    global global_var
    global_var = 20# global copy of variable will be modified

modify_global()
print(global_var)  # Output will be 20
```

The lifetime of a variable:

The lifetime of a variable in Python depends on its scope. Global variables persist throughout the program's execution, whereas, local variables within functions exist only during the function's execution.

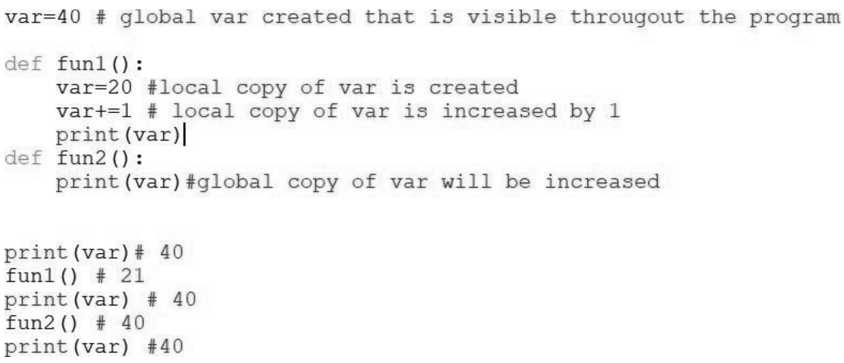
Study the following programs:

Example 1:



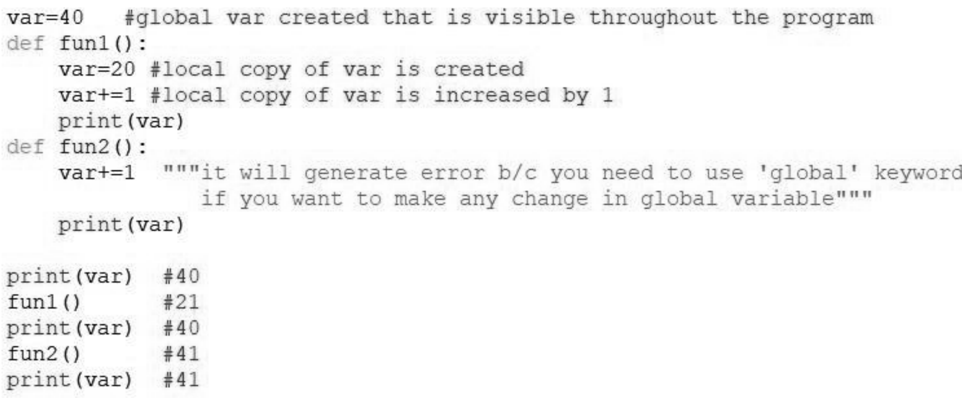
```
g=100 # global variable

def f1():
    '''any changes made to local copy ofvariable g will
     not be refelcetd on global copy of variable g'''

    g=200# local copy of g will ve created, acccessible only in f1()
    g=g+10 # local g will be updated
    print(g)
print(g) # global g is accessible
f1()
print(g) # global variable is accessible
```

Example 2:



```
var=40 # global var created that is visible througout the program

def fun1():
    var=20 #local copy of var is created
    var+=1 # local copy of var is increased by 1
    print(var)
def fun2():
    print(var)#global copy of var will be increased


print(var)# 40
fun1() # 21
print(var) # 40
fun2() # 40
print(var) #40
```

Example 3:



```
var=40   #global var created that is visible throughout the program
def fun1():
    var=20 #local copy of var is created
    var+=1 #local copy of var is increased by 1
    print(var)
def fun2():
    var+=1  """it will generate error b/c you need to use 'global' keyword
                if you want to make any change in global variable"""
    print(var)

print(var)  #40
fun1()      #21
print(var)  #40
fun2()      #41
print(var)  #41
```

Example 4:

```
var=40   #global var created that is visible throughout the program
def fun1():
    var=20 #local copy of var is created
    var+=1 #local copy of var is increased by 1
    print(var)
def fun2():
    global var #global copy of var is accessed
    var+=1  #global copy of var is increased by 1
    print(var)

print(var)  #40
fun1()      #21
print(var)  #40
fun2()      #41
print(var)  #41
```

Passing list as an argument to the function:

Please note that when a list is passed as an argument, the original copy of the list is passed to the function i.e. if any change is made at any index in the list inside the function, it is reflected in the original list. That is because a list is a mutable datatype and in Python, when you pass a list as an argument to a function, you are actually passing a reference to the list rather than a copy of the list. This means that the function parameter will point to the same memory location as the original list. As a result, any changes made to the list within the function will be reflected in the original list outside the function.

```
def modify_list(some_list):
    some_list.append(4)
    some_list[0] = "modified"

my_list = [1, 2, 3]
modify_list(my_list)

print(my_list)  # Output: ['modified', 2, 3, 4]
```

However, if you assign a different list to a variable inside a function in Python, it will create a new local variable that is separate from any variables outside the function. This local variable will only exist within the scope of the function, and changes made to it won't affect the original list outside the function.

```
def modify_list(some_list):
    some_list = [10, 20, 30]  # Assign a new list to the local variable
    print("Inside the function:", some_list)

my_list = [1, 2, 3]
modify_list(my_list)

print("Outside the function:", my_list)
```

Output:

```
Inside the function: [10, 20, 30]
Outside the function: [1, 2, 3]
```

Types of arguments passed to a function:

Positional Arguments:

- These are the most common types of arguments and are matched to the function parameters based on their positions. The first argument corresponds to the first parameter, the second argument corresponds to the second parameter, and so on.
- The number and order of positional arguments must match the function's parameter list.

```
def add(a, b):
    return a + b

result = add(2, 3)  # Here, 2 and 3 are positional arguments.
```

Default Arguments:

- Default arguments are used when a function is called with fewer arguments than there are parameters.
- The default values are specified in the function definition.
- If a value is not provided for a parameter during the function call, the default value is used.

```
def power(base, exponent=2):
    return base ** exponent

result1 = power(3)       # Using the default exponent (2)
result2 = power(3, 4)    # Providing a specific exponent (4)
```

Keyword Arguments:

- In this type, each argument is preceded by a keyword (parameter name) followed by an equal sign.
- The order of keyword arguments does not matter, as they are matched to the function parameters based on their names.
- These arguments provide flexibility to call a function with arguments passed in any order.

```
def add(a, b, c=0):
    return a + b + c

result1 = add(1, 2)
result2 = add(a=1, b=2, c=3)
result3=add(b=10,a=2) # default value of c will be used

print(result1)  # Output: 3
print(result2)  # Output: 6
print(result3)  #output :12
```

Python modules:

- In Python, a module is a file containing Python code that defines variables, functions, and classes.
- Modules allow you to organize and reuse code by breaking it into separate files, making it easier to maintain and understand complex programs.
- Python's standard library comes with a vast collection of built-in modules that cover various functionalities
- If needed, you can also create your own custom modules.
- To use a module in your Python code, you need to import it using the import statement.

math module:

- The math module in Python is a built-in module that provides various mathematical functions and constants.
- It is part of the Python Standard Library i.e. it does not require any additional installation to use.
- To use the math module, you need to import it at the beginning of your Python script.

```
import math
```

- Once you've imported the module, you can access its functions and constants using the math prefix.

Here are some commonly used functions and constants provided by the math module:

Mathematical Constants:

- math.pi: Represents the mathematical constant π (pi).
- math.e: Represents the mathematical constant e (Euler's number).

Basic Mathematical functions :

- math.sqrt(x): Returns the square root of x.
- math.pow(x, y): Returns x raised to the power y.
- math.exp(x): Returns the exponential of x (e^x).
- math.log(x, base): Returns the logarithm of x to the specified base (default base is e).

Trigonometric Functions (all angles are in radians):

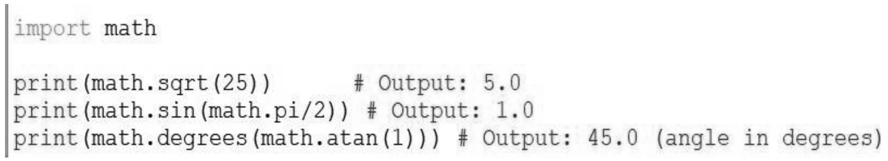
- math.sin(x), math.cos(x), math.tan(x): Sine, cosine, and tangent of x, respectively.
- math.asin(x), math.acos(x), math.atan(x): Arcsine, arccosine, and arctangent of x, respectively.

Hyperbolic Functions:

- math.sinh(x), math.cosh(x), math.tanh(x): Hyperbolic sine, cosine, and tangent of x, respectively. Angular Conversion:
- math.degrees(x): Converts x from radians to degrees.
- math.radians(x): Converts x from degrees to radians. Miscellaneous:
- math.ceil(x): Returns the smallest integer greater than or equal to x.
- math.floor(x): Returns the largest integer less than or equal to x.
- math.factorial(x): Returns the factorial of x.

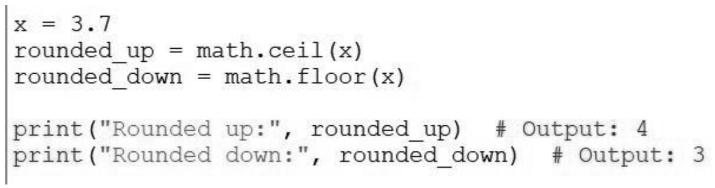
Study the following examples:

Example 1:



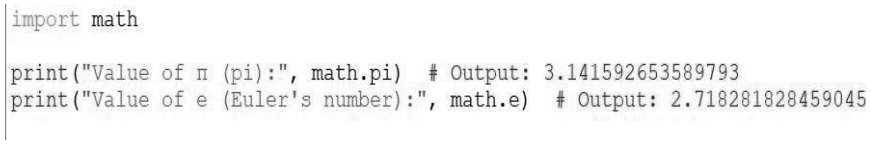
```
import math

print(math.sqrt(25))      # Output: 5.0
print(math.sin(math.pi/2)) # Output: 1.0
print(math.degrees(math.atan(1))) # Output: 45.0 (angle in degrees)
```

Example 2:



```
x = 3.7
rounded_up = math.ceil(x)
rounded_down = math.floor(x)

print("Rounded up:", rounded_up)  # Output: 4
print("Rounded down:", rounded_down)  # Output: 3
```

Example 3:



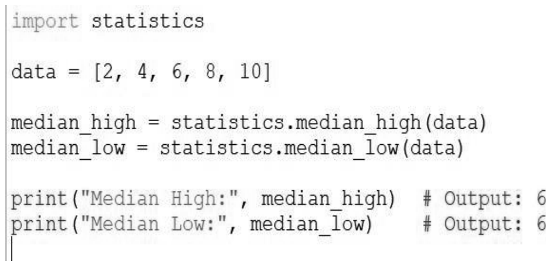
```
import math

print("Value of π (pi):", math.pi)  # Output: 3.141592653589793
print("Value of e (Euler's number):", math.e)  # Output: 2.718281828459045
```

Statistics module:

- The statistics module in Python is another built-in module that provides functions for working with statistical data.
- It offers a variety of statistical functions to compute measures like mean, median, standard deviation, variance, etc.
- The statistics module is part of the Python Standard Library, so there's no need to install any additional packages to use it.

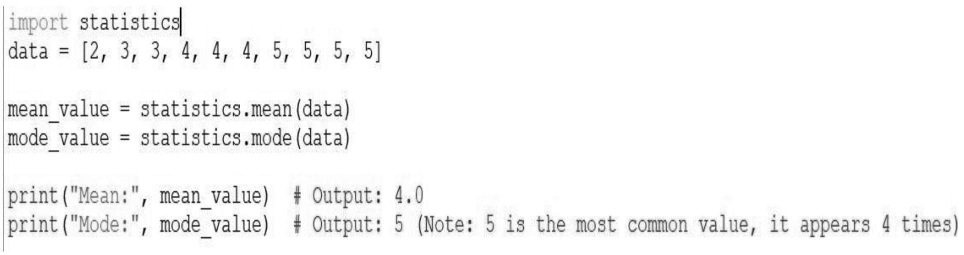
  Here are some commonly used functions provided by the statistics module:
- statistics.mean(data): Calculates the arithmetic mean (average) of the data.
- statistics.median(data): Computes the median value of the data.
- statistics.mode(data): Finds the mode (most common value) in the data.

Example 1:



```
import statistics

data = [2, 4, 6, 8, 10]

median_high = statistics.median_high(data)
median_low = statistics.median_low(data)

print("Median High:", median_high)  # Output: 6
print("Median Low:", median_low)    # Output: 6
```

Example 2:

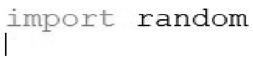


```
import statistics
data = [2, 3, 3, 4, 4, 4, 5, 5, 5, 5]

mean_value = statistics.mean(data)
mode_value = statistics.mode(data)

print("Mean:", mean_value)  # Output: 4.0
print("Mode:", mode_value)  # Output: 5 (Note: 5 is the most common value, it appears 4 times)
```

random module

- The random module in Python is another built-in module that provides functions for generating random numbers, and sequences, and making random choices.

- It is commonly used for tasks such as random number generation, random shuffling, and random sampling.



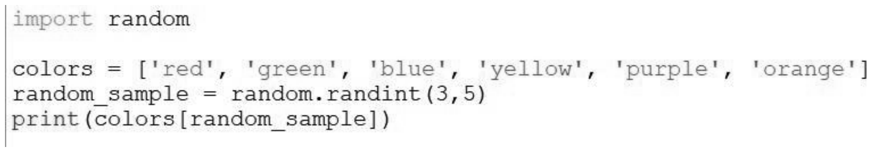
```
import random
```

Here are some commonly used functions provided by the random module:

- random.random(): Generates a random float number in the range [0.0, 1.0). • random.uniform(a, b): Generates a random float number in the range [a, b).
- random.randint(a, b): Generates a random integer in the range [a, b] (inclusive).
- random.choice(sequence): Picks a random element from a sequence (list, tuple, string, etc.).
- random.shuffle(sequence): Shuffles the elements of a sequence randomly (in-place).

Example 1:

What is the possible outcome/s of the following code?



```
import random

colors = ['red', 'green', 'blue', 'yellow', 'purple', 'orange']
random_sample = random.randint(3,5)
print(colors[random_sample])
```

Possible options:

a) green
b) yellow
c) blue
d) orange

Solution:

Here, the possible values for the variable random sample are 3, 4 and 5. Hence, the possible Outputs of the above code are b) Yellow and d) orange.

Example 2:

Code:

```
import random

# Generate a random float between 0 and 1
random_float = random.random()
print("Random float between 0 and 1: ", random_float)

# Generate a random float between a and b
a = 5
b = 10
random_uniform = random.uniform(a, b)
print("Random float between ",a," and ",b, " : ", random_unifor

# Generate a random integer between a and b (inclusive)
a = 1
b = 6
random_int = random.randint(a, b)
print("Random integer between ", a, " and ",b," : ",random_int)

# Generate a random element from a list
fruits = ["apple", "banana", "cherry"]
random_fruit = random.choice(fruits)
print("Random fruit: ",random_fruit)

# Shuffle a list
colors = ["red", "green", "blue", "yellow"]
random.shuffle(colors)
print("Shuffled colors: ", colors )
```

Output:

```
Random float between 0 and 1:  0.14799122104768725
Random float between  5  and  10  :  6.5708962798366475
Random integer between  1  and  6  :  3
Random fruit:  cherry
Shuffled colors:  ['red', 'blue', 'yellow', 'green']
```

Example 3:

What are the possible output/s for the following code?

```
import random
low=25
point =5
for i in range (1,5):
    Number=low + random.randint(0,point)
    print(Number,end=" : ")
    point-=1;
print()
```

Output Options:

i. 29 : 26 : 25 : 28 :   ii. 24 : 28 : 25 : 26 :

iii. 29 : 26 : 24 : 28 :   iv. 29 : 26 : 25 : 26 :

Solution:

Option iv

Example 4:

What are the possible outcome/s for the following code:

```
import random
LIMIT = 4
Points = 100 +random.randint(0,LIMIT);
for p in range(Points,99,-1):
     print(p,end="#")
print()
```

Output Options:

i. 103#102#101#100#   ii. 100#101#102#103#

iii. 100#101#102#103#104#   iv. 4#103#102#101#100#

Solution:

Option i and option iv

Assignment

1. What will be the output of the following code?

```
a=10
def call():
    global a
    b=20
    a=a+b
    print(a)
call( )
```

a) 10   b) 30   c) error   d) 20

2. What is the scope of a variable defined inside a function?

a) Global scope   b) Local scope   c)Universal scope   d)Function scope

3. In Python, can a function return multiple values simultaneously?

a) Yes   b) No

4. What is the purpose of the "return" statement in a function?
   a) It specifies the type of the function.
   b) It defines the input parameters of the function.
   c) It indicates the end of a function.
   d) It returns a value from the function to the caller.

5. Which of the following module functions generates an integer?

   a) randint()   b) uniform()   c) random()   d) all of the above

6. The return type of the input() function is

   a) String   b) integer   c) list   d) tuple