UNIT 3

Value Returning Function

A value-returning function is a function that returns a value back to the part of the program that called it. Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks. These libraries typically contain a function that generates random numbers.

A value-returning function is a special type of function. It is like a simple function in the following ways.

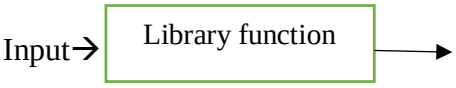
- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

Standard Library Functions and the import Statement:

Standard library of functions that have already been written

import math

A library function viewed as a black box



This statement causes the interpreter to load the contents of the math module into memory and makes all the functions in the math module available to the program.

These modules, which are copied to your computer when you install Python, help organize the standard library functions. For example, functions for performing math operations are stored together in a module, functions for working with files are stored together in another module, and so on.

Generating Random Numbers

Random numbers are useful for lots of different programming tasks. The following are just a few examples.

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.

- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave.

Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.

• Random numbers are useful in statistical programs that must randomly select data for analysis.

• Random numbers are commonly used in computer security to encrypt sensitive data.

```
number = random.randint(1, 100)
```

import Python provides several library functions for working with random numbers. These functions are stored in a module named random in the standard library. To use any of these functions you first need to write this import statement at the top of your program:

```
import random
```

This statement causes the interpreter to load the contents of the random module into memory.

```
number = random.randint(1, 100)
```

arguments

```
def main():
 # Get a random number.
 number = random.randint(1, 10)
 # Display the number.
 print('The number is', number)
 # Call the main function.
 main()
```

Program Output

The number is 7

The randrange, random, and uniform Functions

The randrange function takes the same arguments as the range function. The difference is that the randrange function does not return a list of values. Instead, it returns a randomly selected value from a sequence of values. For example, the following statement assigns a random number in the range of 0 through 9 to the number variable:

```
number = random.randrange(10)
```

The function will return a randomly selected number from the sequence of values 0 up to, but not including, the ending limit. The following statement specifies both a starting value and an ending limit for the sequence:

number = random.randrange(5, 10)

When this statement executes, a random number in the range of 5 through 9 will be assigned to number. The following statement specifies a starting value, an ending limit, and a step value:

number = random.randrange(0, 101, 10)

In this statement the randrange function returns a randomly selected value from the following sequence of numbers:

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Random Number Seeds

They are actually pseudo random numbers that are calculated by a formula. The formula that generates random numbers has to be initialized with a value known as a seed value. The seed value is used in the calculation that returns the next random number in the series. When the random module is imported, it retrieves the system time from the computer's internal clock and uses that as the seed value. The system time is an integer that represents the current date and time, down to a hundredth of a second.

example:

```
random.seed(10)
```

In this example, the value 10 is specified as the seed value. If a program calls the random.seed function, passing the same value as an argument each time it runs, it will always produce the same sequence of pseudorandom numbers.

```
>>> import random e
>>> random.seed(10) e
 >>> random.randint(1, 100) e
      58
 >>> random.randint(1, 100) e
      43
 >>> random.randint(1, 100) e
     58
>>> random.randint(1, 100) e
     21
```

Write your own returning function

A value-returning function has a return statement that returns a value back to the part of the program that called it.

A value-returning function in the same way that you write a simple function, with one exception: a value-returning function must have a return statement.

```
def function_name():
statement
statement
etc.
```

One of the statements in the function must be a return statement, which takes the following form:

```
return expression
```

The value of the expression that follows the key word return will be sent back to the part of the program that called the function. This can be any value, variable, or expression that has a value Here is a simple example of a value-returning function:

```
def sum(num1, num2):
result = num1 + num2
return result


def main():
 # Get the user's age.
 first_age = int(input('Enter your age: '))
# Get the user's best friend's age.
 second_age = int(input("Enter your best friend's age: "))
 # Get the sum of both ages.
total = sum(first_age, second_age)
 # Display the total age.
 print('Together you are', total, 'years old.')
 # The sum function accepts two numeric arguments and
 # returns the sum of those arguments.
 def sum(num1, num2):
 result = num1 + num2
```

```
return result
# Call the main function.
main()
```

Program Output

```
Enter your age: 22 e
Enter your best friend's age: 24 e
Together you are 46 years old.
```

Returning Strings

For example, the following function prompts the user to enter his or her name, and then returns the string that the user entered.

```
def get_name():
# Get the user's name.
name = input('Enter your name: ')
# Return the name.
return name
```

Returning Boolean Values

Python allows you to write Boolean functions, which return either True or False.

```
number = int(input('Enter a number: '))
if (number % 2) == 0:
print('The number is even.')
else:
print('The number is odd.')
```

Returning Multiple Values

In Python, however, you are not limited to returning only one value. You can specify multiple expressions separated by commas after the return statement, as shown in this general format:

```
return expression1, expression2, etc.
```

```
def get_name():
# Get the user's first and last names.
first = input('Enter your first name: ')
```

```
last = input('Enter your last name: ')
# Return both names.
return first, last
```

The Math module:

Math Module Function	Description
acos(x)	Returns the arc cosine of x, in radians
asin(x)	Returns the arc sine of x, in radians
atan(x)	Returns the arc tangent of x, in radians.
ceil(x)	Returns the smallest integer that is greater than or equal to x
cos(x)	Returns the cosine of x in radians.
degrees(x)	Assuming x is an angle in radians, the function returns the angle converted to degrees.
exp(x)	Returns ex
floor(x)	Returns the largest integer that is less than or equal to x.
log(x)	Returns the natural logarithm of x.
radians(x)	Assuming x is an angle in degrees, the function returns the angle converted to radians.
sin(x)	Returns the sine of x in radians
sqrt(x)	Returns the square root of x.
tan(x)	Returns the tangent of x in radians

The Python standard library's math module contains numerous functions that can be used in mathematical calculations.

```
result = math.sqrt(16)
import math
 def main():
 # Get a number.
 number = float(input('Enter a number: '))
 # Get the square root of the number.
```

```
 square_root = math.sqrt(number)
 # Display the square root.
 print('The square root of', number, 'is', square_root)
 # Call the main function.
 main()
```

```
Enter a number: 25 e
The square root of 25.0 is 5.0
```

Storing Function in Module

A module is a file that contains Python code. Large programs are easier to debug and maintain when they are di complex, the need to organize your code becomes greater. You have already learned that a large and complex program should be divided into functions that each performs a specific task.

A module is simply a file that contains Python code. When you break a program into modules, each module should contain functions that perform related tasks.

```
import math
 # The area function accepts a circle's radius as an
 # argument and returns the area of the circle.
 def area(radius):
return math.pi * radius**2
# The circumference function accepts a circle's
# radius and returns the circle's circumference.
 def circumference(radius):
 return 2 * math.pi * radius
```

we should mention the following things about module names:

• A module's file name should end in .py. If the module's file name does not end in .py you will not be able to import it into other programs.

• A module's name cannot be the same as a Python key word. An error would occur, for example, if you named a module for.

To use these modules in a program, you import them with the import statement. Here is an example of how we would import the circle module:

EXCEPTIONS IN PYTHON

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

a)Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

```
# initialize the amount variable
amount = 10000
# check that You are eligible to
# purchase Dsa Self Paced or not
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

Output:

Exceptions: Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
# initialize the amount variable
marks = 10000 26
# perform division with 0
a = marks / 0
print(a)
```

Output:

b) Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Example: Let us try to access the array element whose index is out of bound and handle the corresponding exception.

```
# Python program to handle simple runtime error
#Python 3
a = [1, 2, 3]
try:
print ("Second element = %d" %(a[1]))
# Throws error since there are only 3 elements in array
print ("Fourth element = %d" %(a[3]))
except:
print ("An error occurred")
```

Output:

```
Second element = 2
An error occurred
```

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

c) Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax:

```
try:
# statement(s)
except IndexError:
# statement(s)
except ValueError:
# Statement
```

Example: Catching specific exception in Python

```
# Program to handle multiple errors with one
# except statement
# Python 3
def fun(a):
if a < 4:
# throws ZeroDivisionError for a = 3
b = a/(a-3)
# throws NameError if a >= 4
print("Value of b = ", b)
try:
fun(3)
fun(5)
# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
print("ZeroDivisionError Occurred and Handled")
except NameError:
print("NameError Occurred and Handled")
```

Output

```
ZeroDivisionError Occurred and Handled
If you comment on the line fun(3), the output will be
NameError Occurred and Handled 28
```

d) Try with Else Clause

In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Example: Try with else clause

```
# Program to depict else clause with try-except
# Python 3
# Function which returns a/b
def AbyB(a , b):
try:
c = ((a+b) / (a-b))
except ZeroDivisionError:
print ("a/b result in 0")
else:
print (c)
# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:

```
-5.0
a/b result in 0
```

e) Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception.

Syntax:

```
try:
# Some Code....
except:
# optional block
# Handling of exception (if required)
else: 29 # execute if no exception
finally:
# Some code .....(always executed)
```

Example:

```
# Python program to demonstrate finally
# No exception Exception raised in try block
try:
k = 5//0 # raises divide by zero exception.
print(k)
# handles zero division exception
except ZeroDivisionError:
print("Can't divide by zero")
finally:
# this block is always executed
# regard less of exception generation.
Print ('This is always executed')
```

Output:
Can't divide by zero
This is always executed

f) Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

```
# Program to depict Raising Exception
try:
raise NameError("Hi there") # Raise Error
except NameError:
print ("An exception")
raise # To determine whether the exception was raised or not
```

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line.

```
Traceback (most recent call last):
File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>
raise Name Error("Hi there") # Raise Error
NameError: Hi there
```