Functions

1.3.1 Introduction

A function is a group (or block) of statements that perform a specific task . Functions run only when it is called. One can pass data into the function in the form of parameters. Function can also return data as a result. Instead of writing a large program as one long sequence of instructions , it can be written as several small function , each performing a specific part of the task . They constitute line of code(s) that are executed sequentially from top to bottom by Python interpreter. A python function is written once and is used / called as many time as required . Functions are the most important building blocks for any application in Python and work on the divide and conquer approach. Functions can be conquered into the following three types :

(i) User Defined

(ii) Built in

(iii) Modules

i.User Defined Functions:

Functions should be defined and defined function can be called. Functions are also known as routines, subroutines, methods, procedures or sub-programs. The syntax for defining and calling a function is as illustrated below :

Syntax for Defining a function:

Function is defined using def keyword in Python.

```
def  fun_name(comma_seprated_parameter_ list):
        stmt1
        stmt2
        stmt3
        -------
        stmtn
        return stmt
```

Statements below def begin with four spaces. This is called indentation. It is a requirement of Python that the code following a colon must be indented. A function definition consists of the following components;

1. Keyword def marks the start of function header
2. A function name to uniquely identify it. Function naming follows the same rules as rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function . They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid Python statements that make up the function body. Statements must have same indentation level (usually) 4 spaces)
7. An optional return statement to return a value from the function .

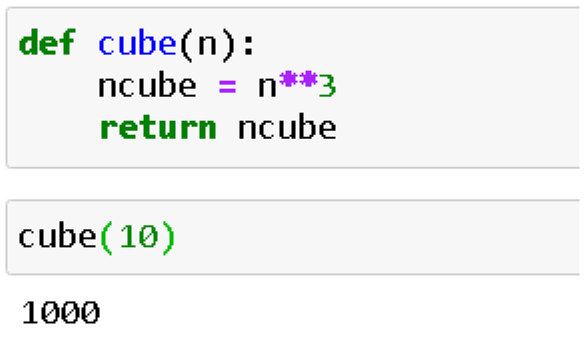
Example :

```
def  cube(n):
     ncube = n**3
     return  ncube
```

Syntax for calling a function:

fun_name(parameter list)

Example: cube(3)



```
def cube(n):
    ncube = n**3
    return ncube
```

```
cube(10)
```

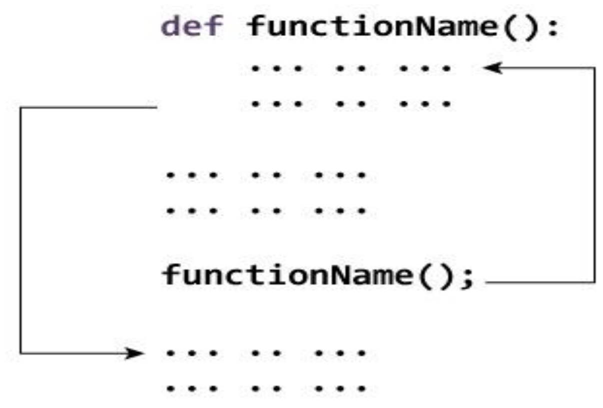
```
1000
```

Parameters and arguments

Parameters are temporary variable names within functions. The argument can be thought of as the value that is assigned to that temporary variable.

- '**n**' here is the ***parameter*** for the function '**cube**'. This means that anywhere we see '**n**' within the function will act as a placeholder until number is passed an argument.
- Here **3** is the ***argument***.
- Parameters are used in ***function definition*** and arguments are used in ***function call***.

Working of function



```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```

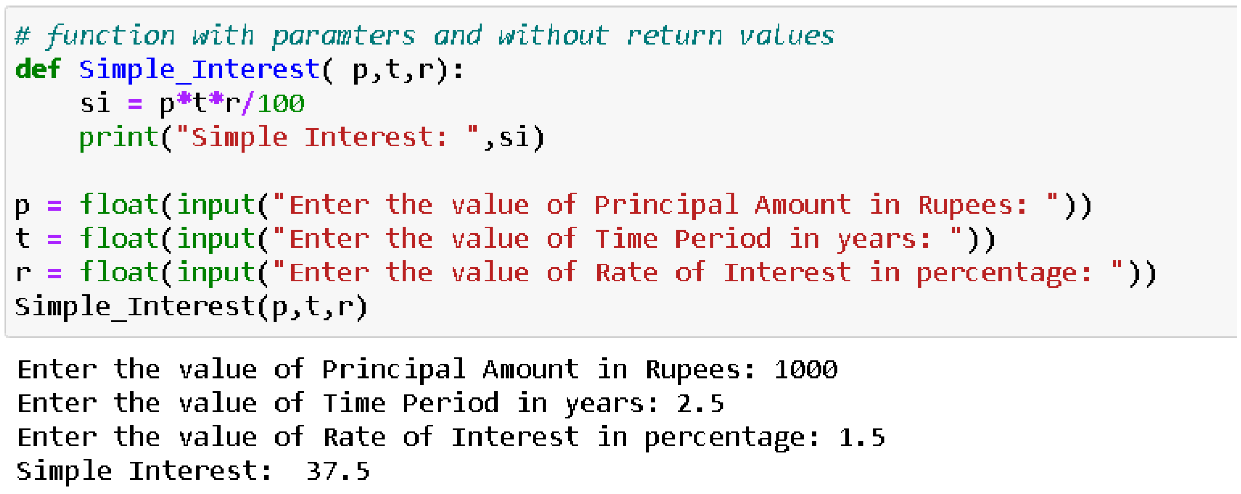
Example 1: Function without parameters

```
# Function without parameters

# Function definition
def  display():
    print("Rocky Robin")
    print("Lucknow")
    print("India")
# Function Call
display()
```

```
Rocky Robin
Lucknow
India
```

Example 2 : Function with parameters but without returning values.

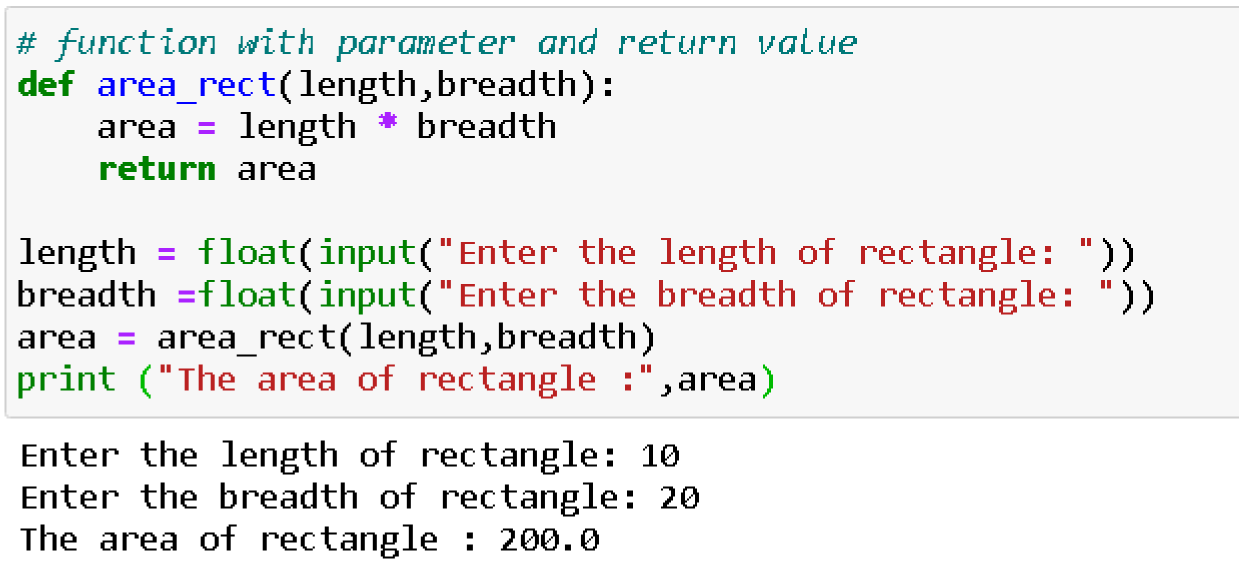


```
# function with paramters and without return values
def Simple_Interest( p,t,r):
    si = p*t*r/100
    print("Simple Interest: ",si)

p = float(input("Enter the value of Principal Amount in Rupees: "))
t = float(input("Enter the value of Time Period in years: "))
r = float(input("Enter the value of Rate of Interest in percentage: "))
Simple_Interest(p,t,r)
```

```
Enter the value of Principal Amount in Rupees: 1000
Enter the value of Time Period in years: 2.5
Enter the value of Rate of Interest in percentage: 1.5
Simple Interest:  37.5
```

Example 3: Function with parameters and return values



```
# function with parameter and return value
def area_rect(length,breadth):
    area = length * breadth
    return area

length = float(input("Enter the length of rectangle: "))
breadth =float(input("Enter the breadth of rectangle: "))
area = area_rect(length,breadth)
print ("The area of rectangle :",area)
```

```
Enter the length of rectangle: 10
Enter the breadth of rectangle: 20
The area of rectangle : 200.0
```

Example 4: Function which return multiple values

```python
def multireturn(n):
    return n**2,n**3,n**4,n**5  # Returning four values
```

```python
n = int(input("Enter the number: "))
np2,np3,np4,np5 = multireturn(n)
print("np2 : ",np2)
print("np3 : ",np3)
print("np4 : ",np4)
print("np5 : ",np5)
```

```
Enter the number: 2
np2 :  4
np3 :  8
np4 :  16
np5 :  32
```

The None-Value

In Python there is a value called None, which represents the absence of a value. None is the only value of the None Type data type. (Other programming languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, None must be typed with a capital N.

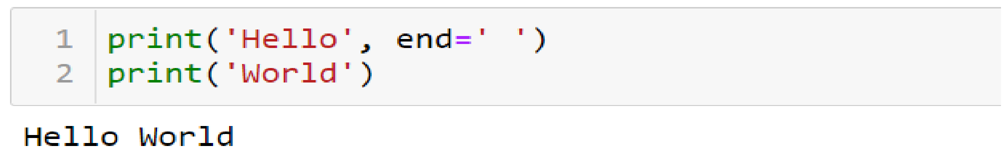
```python
def f1():
    print(1)

x= f1()
y = print("End")
print(x)
print(y)
```

```
1
End
None
None
```

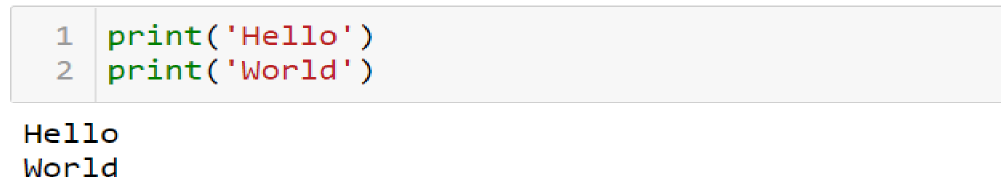
Keyword Arguments and print()

Keyword arguments are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the print( ) function has the optional parameters ***end*** and ***sep*** to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively. Following examples illustrates the behavior of print with ***end*** , without ***end*** and with ***sep***.



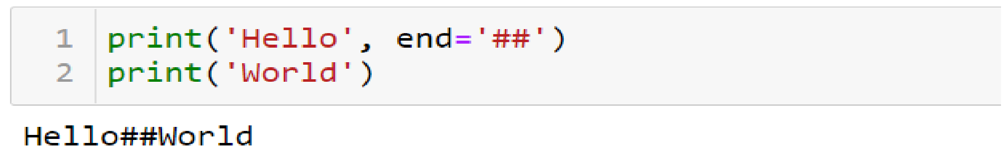
```python
print('Hello', end=' ')
print('World')
```

```
Hello World
```



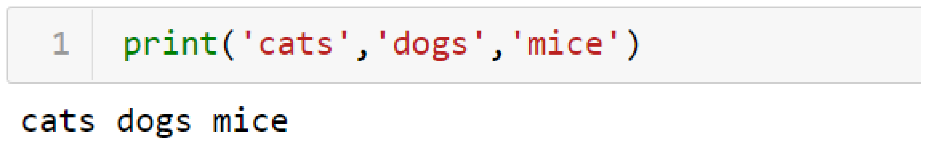
```python
print('Hello')
print('World')
```

```
Hello
World
```



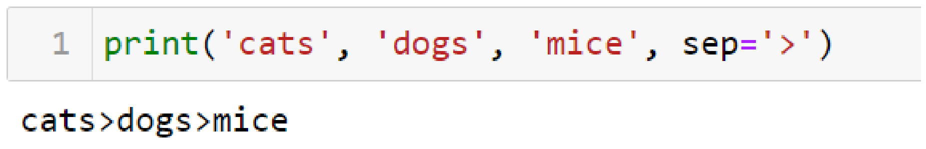
```python
print('Hello', end='##')
print('World')
```

```
Hello##World
```



```python
print('cats','dogs','mice')
```

```
cats dogs mice
```



```python
print('cats', 'dogs', 'mice', sep='>')
```

```
cats>dogs>mice
```

1.3.6 Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global. Following example illustrates the difference between local and global variable .

```
x = 20   # global variable
def f1():
    x = 15
    l = 20 # local variable
    print("Function f1",x,l)

def f2():
    y = x + 20
    print("Function f2",x,y)
```

```
f1()
f2()
print(x)
```

```
Function f1 15 20
Function f2 20 40
20
```

Local variable cannot be used in the global scope

Consider this program which will cause an error when you run it:

```
def fun1():
    x =31337
fun1()
print(x)
```

If you run this program the output will look like this

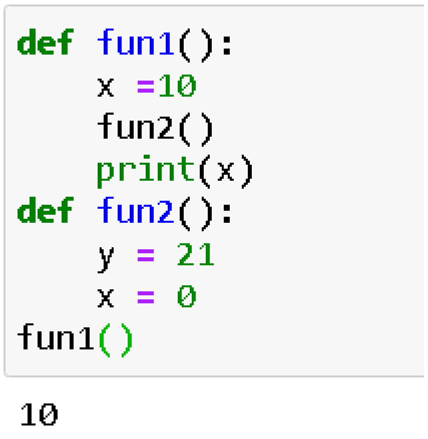
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-8e05a42defd6> in <module>
      2     x =31337
      3 fun1()
----> 4 print(x)

NameError: name 'x' is not defined
```

The error happens because the x variable exists only in the local scope created when fun1() is called. Once the program execution returns from fun1(), that local scope is destroyed, and there is no longer a variable named x. So when your program tries to run print(x), Python gives you an error saying that x is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can’t be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:



```
def fun1():
    x =10
    fun2()
    print(x)
def fun2():
    y = 21
    x = 0
fun1()
```

```
10
```

When the program starts the func1() is called and a local scope is created . The local variable x is set to 10. Then fun2() is called and a second local scope is created . Multiple local scopes can exist at the same time . In this new local scope , the local variable y is set to 21 and a local variable x which is different from the one in fun1()’s local scope is also created and set to 0. When fun2() returns the local scope for the call to fun1() still exists here the x variable is set to 10. This is what the programs prints.

The local variables in one function are completely separate the local variable in another function.

Global Variable Can be read from a local scope :

Consider the following program :

```python
def fun1():
    print(x)
x = 42
fun1()
print(x)
```

```
42
42
```

Since there is no parameter named x or any code that assigns x a value in the fun1() function , when x is used in fun1(), Python considers it a reference to the global variable x. This is why 42 is printed when the previous program is run.

Local and Global Variables with the same Name :

One should avoid using local variables that have the same name as a global variable or another local variable. Consider the following program :

```python
def fun1():
    x = 'I am local to fun1'
    print(x)
def fun2():
    x = 'I am local to fun2'
    print(x)
    fun1()
    print(x)
x = 'I am global'
fun2()
print(x)
```

When you run the program , it outputs the following :

```
I am local to fun2
I am local to fun1
I am local to fun2
I am global
```

There are actually three different variables in this program, but confusinglgl they are all named x.

A variable named x that exists in a local scope when fun1() is called.

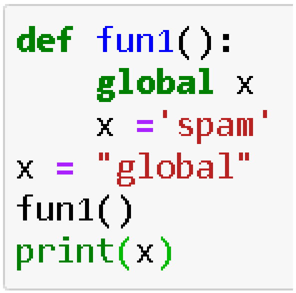
A variable named x that exists in a local scope when fun2() is called

A variable named x that exists in the global scope

Since these three separate variables all have the same name, it can be confusing to keep of which one is being used at any given time . This is why you should avoid using the same variable name in different scopes.

Global Statement

If you need to modify a global variable from within a function , used the global statement . If you have a line such as global x at the top of a function , it tells Python, In this function, x refers to the global variable, so don't create a local variable with this name." For example, type the following code and run



```
def fun1():
    global x
    x ='spam'
x = "global"
fun1()
print(x)
```

When you run this program the final print() call will output this :

Because x is declared global at the top of spam() , when x is set to 'spam' , this assignment is done to the globally scoped x. No local x variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.

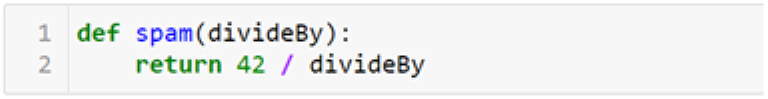
2. If there is a global statement for that variable in a function, it is a global variable.

3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.

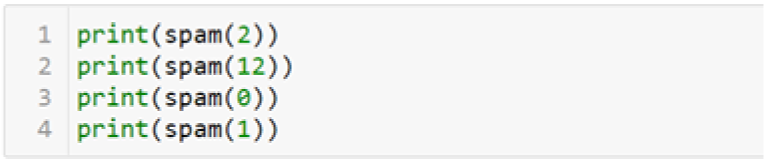
4. But if the variable is not used in an assignment statement, it is a global variable.

.Exceptional Handling

Right now, getting an error, or exception, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run. For example, consider the following program, which has a "divide-byzero" error. Open a new file editor window and enter the following code, saving it as zeroDivide.py:

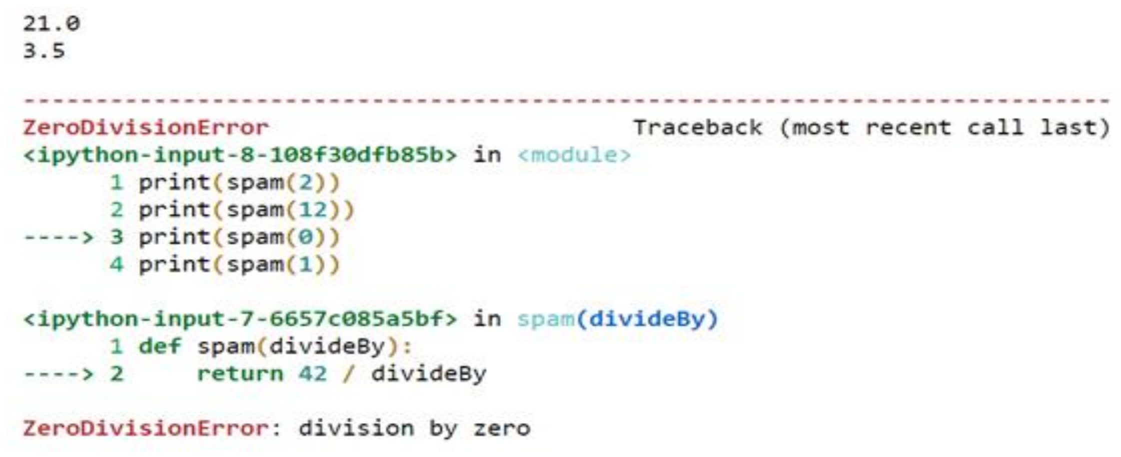


```python
def spam(divideBy):
    return 42 / divideBy
```



```python
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:



```
21.0
3.5

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-108f30dfb85b> in <module>
      1 print(spam(2))
      2 print(spam(12))
----> 3 print(spam(0))
      4 print(spam(1))

<ipython-input-7-6657c085a5bf> in spam(divideBy)
      1 def spam(divideBy):
----> 2     return 42 / divideBy

ZeroDivisionError: division by zero
```

A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.

Try and Except :

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens. You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```python
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
         print('Error: Invalid argument.')
```

```python
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

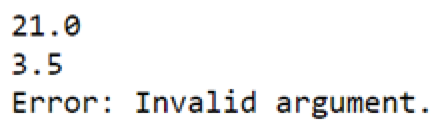
When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```python
# Consider the following program, which instead has the spam()
# calls in the try block:
def spam(divideBy):
    return 42 / divideBy
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:



```
21.0
3.5
Error: Invalid argument.
```

The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down as normal.

ii. Built in functions

Built in functions are the predefined functions that are already available in python. Functions provide efficieincy and structure to a programming language . python has many useful built in functions to make programming easier , faster and more powerful.

Some of the important builtin functions are listed below :

abs()	Returns the absolute value of a number
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value

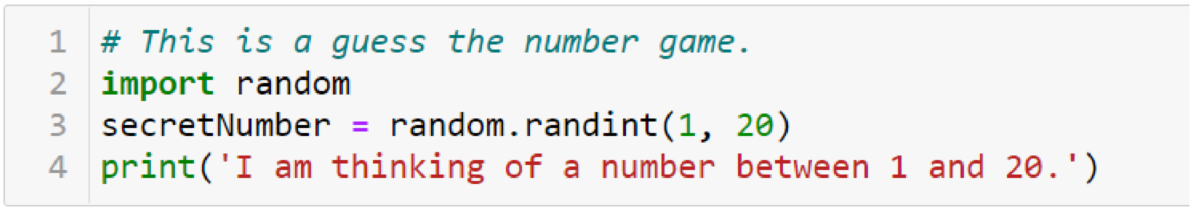
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list

str()	Returns a string object
sum()	Sums the items of an iterator
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__ property of an object
zip()	Returns an iterator, from two or more iterators

iii. Modules :

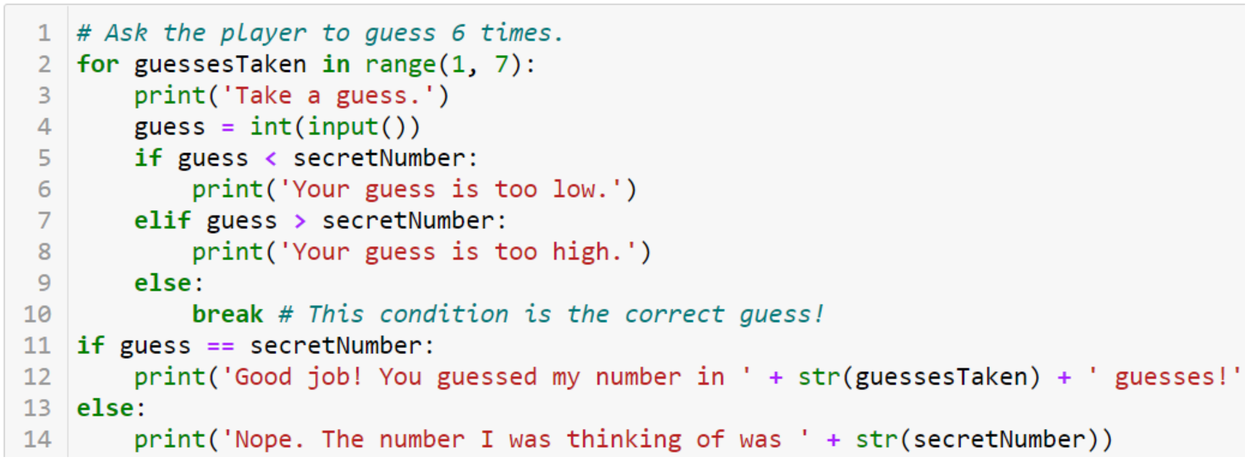
Sample Programs

Example1: A Short Program , Guess the Number



```python
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')
```

```
I am thinking of a number between 1 and 20.
```



```python
# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!'
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

```
Take a guess.
8
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
13
Your guess is too low.
Take a guess.
15
Your guess is too high.
Take a guess.
14
Good job! You guessed my number in 5 guesses!
```

Project:

```python
def collatz(n):
    if n%2==0:
        print(n//2)
        return n//2
    elif n%2==1:
        print(3*n+1)
        return 3*n+1
```

```python
while True:
    try:
        n = int(input("enter a number "))
    except ValueError:
            print("Invalid Argument")
    x = collatz(n)
    if x==1:
        break
```

```
enter a number 5
16
enter a number 3
10
enter a number 4
2
enter a number 2
1
```