# FUNCTIONS

In [8]:
```
import math

def root(num):
    result = math.sqrt(num)
    return result

root(5)
```

Out[8]: 2.23606797749979

In [6]:
```
def greatest(num1, num2):
    if num1 > num2:
        return True
    return False

greatest(5,8)
```

Out[6]: False

In [ ]:
```
# A **function** is a named block of code. Before now, we have been using some Python built-in functions like p
To **define** a function, we need to specify its **name**, **parameters** (which are optional) and **block of c
Let's write an example of a user-defined function that returns True if a number is even, otherwise it returns F
```

In [12]:
```
def is_even(number):
    if(number % 2 == 0):
        return True
    return False
```

As seen in the example above, we used the **def** keyword to define a function. A function can have parameters. A **parameter** is a variable name specified within the **parenthesis** in a **function definition**. Functions may or may not return a value, the **return** keyword is used to make a function return a result. Values returned by functions can be saved to a variable for later use.
To use a function, you have to **call** it by its **name** and pass the necessary **arguments** if any is required. An **argument** is a value passed to the **function's parameters** when the function is called.

In [ ]:
```
print(is_even(8))
```

In [ ]:
```
is_even(5)
```

As seen above, the function is_even() was called with 8 and 5 as arguments.

We can also have functions that do not return any value. Such functions might just be used to display an output, perform some calculations or modify some variables. Let's see an example of a function that does not return a value but just prints even numbers between 1 and a specified number (inclusive).

In [14]:
```
def even_numbers(n):
    if (n <= 1):
        print(n,'should be greater than 1.')
    else:
        for num in range(1, n+1):
            if(is_even(num)):
                print (num)
```

In [15]:
```
even_numbers(1)
```

```
1 should be greater than 1.
```

In [16]:
```
even_numbers(5)
```

```
2
4
```

For functions in which no value is returned using the `return keyword`, Python returns a `None` value as default. In the example below, the function `double` only doubles the argument and doesn't return any value.

In [17]:
```
result = is_even(12)
print(result)
```

```
True
```

In [27]:
```
def double(n):
    result =n*2
    return result
```

In [ ]:
```

result = double(5)
print(result)
```

# DEFAULT PARAMETER VALUES

Function parameters can be assigned a default value during the function definition so that if the function is called without an argument, the default parameter value will be used. In a function definition, parameters with default values (optional parameters) should come after the required parameters.

In [28]:
```
def best_food(food='Chicken and chipsa'):
    print('My best food is', food)
```

In [30]:
```
best_food("yam")
```

```
My best food is yam
```

In [31]:
```
best_food('Indomie and Egg')
```

```
My best food is Indomie and Egg
```

In [32]:
```
def about_me(name, food='Chicken and chips'):
    print('My name is', name)
    print('My best food is', food)
```

In [33]:
```
about_me('Aminat')
```

```
My name is Aminat
My best food is Chicken and chips
```

In [34]:
```
about_me('Aminat', 'Chicken Salad')
```

```
My name is Aminat
My best food is Chicken Salad
```

# KEYWORD ARGUMENTS

Arguments can be passed to functions using the **key = value syntax**. When using this syntax, the order in which the arguments are passed doesnt matter. Let's see an example of a function call with and without the key value syntax.

In [35]:
```
def person(name, gender, age):
    print ('My name is', name)
    print('I am a', gender)
    print('I am',age,'years old.')
```

In [36]:
```
person('Damola', 'male', 24)
```

```
My name is Damola
I am a male
I am 24 years old.
```

In [37]:
```
person(gender='female', age=15 , name='Esther')
```

```
My name is Esther
I am a female
I am 15 years old.
```

# ARBITRARY ARGUMENTS

A function can take a variable list of arguments. To define a function that takes any number of arguments, we use `*` before the parameter name in the function definition. This `*` tells Python to gather all the arguments into a tuple (a datastructure we would discuss later) of the named parameter.

In [38]:
```
def Addition (*numbers):
    total = 0
    for number in numbers:
        total+= number
    return total
```

In [39]:
```
print(Addition())
```

```
0
```

In [40]:
```
print(Addition(3,5,7))
```

```
15
```

In [41]:
```
def Add (*args):
    return sum(args)

Add(4,5,66)
```

Out[41]: 75

# RECURSSIVE FUNCTIONS

A recursive function is a function that calls itself within its function definition. It is a mathematical and programming concept that loops through data to attain a result. Recurssion is a very effective approach to programming if it is written very well.

In [42]:
```
def factorial(n): #using iteration
    if (n == 0 or n == 1):
        return 1
    answer = 1
    for i in range(n,0,-1):
        answer*=i
    return answer

#calling the function
factorial(5)
```

Out[42]: 120

In [43]:
```
def recursive_factorial(n):
    if n == 0 or n == 1: #base case
        return 1
    ans = n * recursive_factorial(n-1)  #recursive case
    return ans
```

In [44]:
```
recursive_factorial(4)
```

Out[44]: 24

In [ ]:
```
def recursive_fibonacci (n):
    if (n == 1 or n == 0): #base case
        return 1
    ans = recursive_fibonacci(n-2) + recursive_fibonacci(n-1)#recursive case
    return ans

recursive_fibonacci(5)
```

# SCOPE

A variable is only available from inside the region it is created. This is called scope. We have 2 types of scope:

1. **Local scope:** A variable created inside a function belongs to the local scope of that function and can only be used inside that function.
2. **Global scope:** A variable created outside a function or within the main part of a Python code belongs to a global scope. It is available for use both locally and globally.

In [4]:
```
# global variables
name = 'Amanda'
age = 20
nationality = 'Nigerian'

def person(name, age):
    #using the name and age local variables and the nationality global variable
    print(f"My name is {name}. I am a {nationality}. I am a {age} years old.")
```

In [5]:
```
person('Brian', 25)
```

```
My name is Brian. I am a Nigerian. I am a 25 years old.
```

In [3]:
```
#Accessing the global variables
print(f"My name is {name}. I am a {nationality}. I am {age} years old.")
```

```
My name is Amanda. I am a Nigerian. I am 20 years old.
```

# THE GLOBAL KEYWORD

We can create or modify a global variable within a function using the global keyword. if we do not put the global keyword in the function before calling or using the variable, Python will create the variable as a new local variable. Lets see an example.

In [10]:
```
nationality = 'Nigerian'
def person(name, age):
    global nationality
    print(f"My name is {name}. I am a {nationality}. I am a {age} years old.")
```

In [9]:
```
person('Brian', 25)
print(nationality)
```

```
My name is Brian. I am a Nigerian. I am a 25 years old.
Nigerian
```

In [11]:
```
a = 5
b = 10
c = 4
def multiply(a, b):
    #locally created variable c
    c = 7
    print('Local C value ',c)
    return a * b * c

print(multiply(2,4))

print ('Global C Value',c)
```

```
Local C value  7
56
Global C Value 4
```

In [11]:
```
a = 5
b = 10
c = 4

def multiply(a, b):
    global c
    c = 10
    print('C value ',c)
    return a * b * c

print(multiply(2,4))

print ('Global C Value',c)
```

```
C value  10
80
Global C Value 10
```

# EXERCISE: PRIME NUMBERS

Write a function that prints all prime numbers up to a specified number.

In [3]:
```
def primenumbers(n):
    for num in range (1, 101):
        for i in range(2, num):
            if num % i == 0:
                break
        else:
            print(n)
    primenumber(30)
```

In [ ]:
```

```