User Defined Functions - Explained

April 17, 2023

1 User Defined Functions - Explained

1.1 Function Definition and Function Call

1.2 Function Definition

- If you want to create a user defined function we have to define it first.
- In Python functions can be defined using def keyword
- Syntax for function definition

```
def function_name(parameters): # parameters are optional
    function body
```

- Here **function_name** is any valid identifier name in python
- **parameters** are the inputs that this function takes (if any), you can also choose not to define any parameters
- **function body** is the task that the function is intened to perform

1.3 Function Call

- Creating a function is not enough. If you want to use the function, you have to call it.
- Syntax for a function call

```
function_name(arguments) # here arguments are optional
```

- You have to make a function call appropriate with the definition.

1.4 Types of User Defined Functions

- Functions without arguments without return type
- Functions with arguments without return type
- Functions with arguments with return type (Mostly used)
- Functions without arguments with return type

1.4.1 Functions without arguments without return type

- These functions will not take any arguments and will not return any values to the caller function
- Syntax for creating these type of functions in python

```
def function_name():
    body of the function
```

- And functions like this can be called directly.
- Syntax for function call

```
function_name()
```

- Examples

```
# creating the function
def say_hi():
    print('Hi this is Python 3.10 wishing you!')
# This function say_hi() can be called directly in the program as shown below.
say_hi() # It prints Hi this is Python 3.10 wishing you!, everytime it's called
```

- **Things to observe**
- Above created function takes no arguments. So we don't need to pass any while calling.
- Above function doesn't return anything to the caller function. So we don't need to assign it to a variable.
- Above function contains a print() statement within the function body. So we don't need to put it inside a print() statement while calling

[1]:
```
# Function definition
def say_hi():
    print('Hi this is Python 3.10 wishing you!')

# Function call(s)
say_hi()
say_hi()
say_hi()
```

```
Hi this is Python 3.10 wishing you!
Hi this is Python 3.10 wishing you!
Hi this is Python 3.10 wishing you!
```

[11]:
```
# Function Definition
def wish_them_all():
    print("Hello all, howdy!!!")

# Function call
wish_them_all() # calling directly
```

```
Hello all, howdy!!!
```

[79]:
```
# A simple function that shows Temporarily out of Service message when called
# Function Definition
def OOS():
    print('Sorry! Temporarily out of service')

# Function call(s)
OOS()
OOS()
```

```
Sorry! Temporarily out of service
Sorry! Temporarily out of service
```

[80]:
```
# Function that show SOS message upon calling
# Function definition
def SOS():
    print('SAVE OUR SOULS! SAVE OUR SOULS! SAVE OUR SOULS!')

# Function call
SOS()
```

```
SAVE OUR SOULS! SAVE OUR SOULS! SAVE OUR SOULS!
```

[ ]:
```

```

1.4.2 Functions with arguments and without return type

- Giving variables inside the parenthese while defining a function make the function as parameterized function
- Syntax to create a parameterized function

```
def function_name(param1, param2, param3, ...., paramn):
    function body
```

- In the above function **param1, param2, param3, ..., paramn** are parameters to the function, which will be passed by user as arguments while function calling.
- We will do something inside the function body using those parameters
- Example

```
def add(num1, num2): # parameters are num1 and num2
    result = num1 + num2 # using them in function body
    print(result) # printing result
```

- Here **num1 and num2** are parameters (inputs) to the function add(), which prints their sum as result upon calling
- **Things to observe**
- Above created function add(num1, num2) is a paramerized function, so while calling the function, we have to pass sufficient (right amount) of arguments as inputs
- **Valid function calls of the above function add(num1, num2) would be**

```
add(10, 20) # Here we are passing exactly 2 arguments (10, 20) to add function as it's exp

a = int(input())
b = int(input())
add(a, b) # This is also a valid function call of add function

add(100, 200) # Also valid
```

- **Invalid function calls of the above function add(num1, num2) would be**

```
add(10, 20, 30) # Here we are passing 3 arguments 10, 20, 30 to add function, but it's only ex
```

```
x = int(input())
add(x) # This is also invalid as we are only passing 1 argument

add() # This is also invalid as we are not passing any agruments at all to a function expecting
```

[32]:
```
# Function definition
def add(num1, num2): # Function with two parameters
    result = num1 + num2 # adding them
    print(result) # printing result (not returning hence without return type function)

# Function call
add(10, 20) # valid function call

x = int(input())
y = int(input())
add(x, y) # yet another valid function call
```

```
30
100
200
300
```

[33]:
```
# Function definition
def add(num1, num2): # Function with two parameters
    result = num1 + num2 # adding them
    print(result) # printing result (not returning hence without return type function)

# Function call
add(10, 20, 30) # invalid function call as add takes only two arguments
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [33], in <cell line: 7>()
      4     print(result) # printing result (not returning hence without return type function)
      6 # Function call
----> 7 add(10, 20, 30)

TypeError: add() takes 2 positional arguments but 3 were given
```

[34]:
```
# Function definition
def add(num1, num2): # Function with two parameters
    result = num1 + num2 # adding them
    print(result) # printing result (not returning hence without return type function)
```

```
# Function call
add(10) # invalid function call as add takes exactly two arguments
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [34], in <cell line: 7>()
      4     print(result) # printing result (not returning hence without return
 ↪type function)
      6 # Function call
----> 7 add(10)

TypeError: add() missing 1 required positional argument: 'num2'
```

[35]:
```
# Function definition
def add(num1, num2): # Function with two parameters
    result = num1 + num2 # adding them
    print(result) # printing result (not returning hence without return type␣
 ↪function)

# Function call
add() # invalid function call as add takes exactly two arguments
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [35], in <cell line: 7>()
      4     print(result) # printing result (not returning hence without return
 ↪type function)
      6 # Function call
----> 7 add()

TypeError: add() missing 2 required positional arguments: 'num1' and 'num2'
```

[ ]:
```

```

Some Examples for functions with arguments and without return type

Let's consider the below task Let's suppose you have print all numbers in the given range (from start to stop) for multiple times (more than once) in the program.

You'd have to write to loop over multiple start and stop values for multiple times as shown below

[46]:
```
# Suppose if you have these values in your program
a = 10
b = 20
```

```
c = 100
d = 200
e = 500
f = 600
# And assume that you have to print all numbers from a to b, from c to d, and also from e to f
# You have to write the following lines of code

for i in range(a, b + 1):  # This is to print numbers from a to b
    print(i, end=' ')

print()
for i in range(c, d + 1):  # This is to print numbers from c to d
    print(i, end=' ')

print()
for i in range(e, f + 1):  # This is to print numbers from e to f
    print(i, end=' ')

# But here you had to WRITE for loop for 3 TIMES to perform a similar type of task
```

```
10 11 12 13 14 15 16 17 18 19 20
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
200
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519
520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579
580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600
```

[48]:
```
# How can we do the same thing using functions with less effort?
# Let's see
# Creating a function called start_to_stop(start, stop) that takes two arguments and prints all
# the numbers from start to stop
def start_to_stop(start, stop): # Function with parameters and without return type
    for i in range(start, stop + 1):
        print(i, end = ' ')
a = 10
b = 20
```

```
c = 100
d = 200
e = 500
f = 600
start_to_stop(a, b) # Function call 1
print()
start_to_stop(c, d) # Function call 2
print()
start_to_stop(e, f) # Function call 3
```

```
10 11 12 13 14 15 16 17 18 19 20
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
200
500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519
520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559
560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579
580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600
```

Let's have a look at another task that illustrates the use of functions Suppose that you have 5 different subjects marks (out of 100) of 3 students with you and you have to grade them based on the percentage they've got out of 100.

Percentage Calculation: obtained marks / total marks x 100

student1 = [58, 66, 32, 70, 36]
student2 = [75, 58, 55, 95, 83]
student3 = [75, 86, 82, 90, 93]

And grading criteria is as follows

Grading Criteria

Percentage Range	Grade
percentage>=90	O
80<=percentage<90	A
70<=percentage<80	B
60<=percentage<70	C
50<=percentage<60	D
35<=percentage<50	E
percentage<35	F

[77]:
```
# The normal (non functional) approach to perform the above task would be as␣
 ↪follows
student1 = [58, 66, 32, 70, 36]
student2 = [75, 58, 55, 95, 83]
student3 = [75, 86, 82, 90, 93]

s1_per = sum(student1) / 500 * 100
if s1_per >= 90:
    print('O')
elif s1_per >= 80:
    print('A')
elif s1_per >= 70:
    print('B')
elif s1_per >= 60:
    print('c')
elif s1_per >= 50:
    print('D')
elif s1_per >= 35 and s1_per <50:
    print('E')
else:
    print('F')

s2_per = sum(student2) / 500 * 100
if s2_per >= 90:
    print('O')
elif s2_per >= 80:
    print('A')
elif s2_per >= 70:
    print('B')
elif s2_per >= 60:
    print('c')
elif s2_per >= 50:
    print('D')
elif s2_per >= 35 and s2_per <50:
    print('E')
else:
    print('F')

s3_per = sum(student3) / 500 * 100
if s3_per >= 90:
    print('O')
elif s3_per >= 80:
    print('A')
elif s3_per >= 70:
    print('B')
elif s3_per >= 60:
    print('c')
```

```
elif s3_per >= 50:
    print('D')
elif s3_per >= 35 and s3_per <50:
    print('E')
else:
    print('F')
```

```
D
B
A
```

[78]:
```
# But as you can see above you've done the same thing (finding out grade based on percentage) 3 times. But you wrote code on
# your own for all 3 times
# Let's see how we can use fuctions effecitvely in this task

# Creating a function named set_grade(percentage) which takes percentage as an arguments and print grade based on the percentage
def set_grade(percentage):
    if percentage >= 90:
        print('O')
    elif percentage >= 80:
        print('A')
    elif percentage >= 70:
        print('B')
    elif percentage >= 60:
        print('c')
    elif percentage >= 50:
        print('D')
    elif percentage >= 35 and percentage <50:
        print('E')
    else:
        print('F')

student1 = [58, 66, 32, 70, 36]
student2 = [75, 58, 55, 95, 83]
student3 = [75, 86, 82, 90, 93]

s1_per = sum(student1) / 500 * 100
set_grade(s1_per) # Function call 1

s2_per = sum(student2) / 500 * 100
set_grade(s2_per) # Function call 2

s3_per = sum(student3) / 500 * 100
set_grade(s3_per)
```

```
D
```

```
B
A
```

1.4.3 Functions with argument(s) and with return type (MOSTLY USED FUNCTIONS)

- So far the functions we looked at don't return anything back to the calling function. They just print something inside themselves.
- But a function with a return value is necessay in a lot of occassions as we want to do something with the returned value further down in the program
- Functions with return type will always have a return statement inside the function body
- When a return statement is encountered the function call gets terminated.
- Syntax for creating such function

```
def function_name(parameters):
    process
    return something
```

- Example

```
def add(num1, num2):
    return num1 + num2
```

- The above function **add(num1, num2)** takes two arguments into num1 and num2 and add them and returns the result back to the calling function
- AS THE FUNCTION IS RETURNING SOMETHING BACK TO THE CALLING FUNCTION WE HAVE TO
  - **EITHER STORE THE RETURNED VALUE IN SOME VARIABLE TO USE LATER**
  - **OR HAVE TO CALL THE FUNCTION INSIDE A PRINT STATEMENT TO PRINT IT ON THE SCREEN AS RECEIVED**
- Example(s) for the function call of above function add(num1, num2) would be like

```
s = add(10, 20) # Here we are assigning the function call to a variable so that the returned v
print(s * s) # Using the return value to make some operations further down in the program
```

- Example function call for the same function

```
print(add(100, 200)) # Here, we are making the function call directly in the print statement s
```

Some examples for the function with arguments and with return type

[115]:
```
# Function that takes a number and RETURNS it's reverse
def reverse(num):
    rev = 0
    while num: # Finding out the reverse of num
        r = num % 10
        rev = rev * 10 + r
        num = num // 10
    return rev # Returning the reverse

# Funtion call
```

```
a = reverse(123) # Storing the returned value inside a variable so that we can␣
 ↪use it later
print(a * a)
# or
# Function call 2
print(reverse(987654321)) # Making a function call directly inside a print␣
 ↪statement so that the
# returned value will be printed onto the screen
```

```
103041
123456789
```

HOW TO FIND OUT IF THE FUNCTION IS HAVING A RETURN VALUE ???

- Well, by looking at the function call we can say if the function is having a return value or not
- WE CAN SAY A FUNCTION IS HAVING A RETURN VALUE IF
  - The function call is assigned to a varialbe (Ex: sum = add(10, 20))
  - The funciton call is put directly inside a print statement (Ex: print(add(10, 20))
  - The function call is made inside any other function (built_in or user defined) (Ex: max(add(10, 20), add(100, -90)))
  - The function call is used inside a conditional or a loop as condition (Ex: if add(10, 20) == 30)

[127]:
```
# Creating a function based program to check if a given number is palindrome
# First palindrom number program internally has a process hidden in it, which␣
 ↪is reversing the number
# for which we can definitely write a seperate function

def reverse(number): # defining a function that returns the reverse of the␣
 ↪given number
    rev = 0
    while number:
        r = number % 10
        rev = rev * 10 + r
        number //= 10
    return rev

# Original code
n = int(input("Enter a number: "))
# Using function call inside a condition
if n == reverse(n):
    print(f'{n} is a Palindrome')
else:
    print(f'{n} is not a Palindrome')
```

```
Enter a number: 121
121 is a Palindrome
```

[113]:
```
x = 10
x
```

[113]: 10

[136]:
```
# Printing all palindromes in a range using a return function
# Let's cut this program into 3 parts
# part 1 -> function for reversing a number (with return value (integer))
# part 2 -> another function for checking for palindrome (also with return value (boolean))
# part 3 -> using these functions effectively to get the job done

# Function 1 - Reverse
def reverse(number):
    rev = 0
    while number:
        r = number % 10
        rev = rev * 10 + r
        number = number // 10
    return rev

# Function 2 - Palindrome
# This function is a boolean fuction it returns True if given number is palindrome
# Else it returns False
def is_palindrome(num):  # num = 101
    if num == reverse(num): # Making a function call to funcion 1 (reverse(number))
        return True
    else:
        return False

for i in range(101, 199):
    if is_palindrome(i) == True:
        print(i, end = ' ')
```

```
101 111 121 131 141 151 161 171 181 191
```

[10]:
```
# Function that RETURNS the proper factor sum of a given number
def factor_sum(number): # function definition
    fs = 0
    for i in range(1, number):
        if number % i == 0: # finding factor for number
            fs += i
    return fs

# Making different function calls
```

```
# Function call 1
print(factor_sum(10))

# Function call 2
result = factor_sum(25)
print(result)

# Function call 3
if factor_sum(8) == 7:
    print('YES')
else:
    print('NO')

# Function call 4
print(len(str(factor_sum(10000))))
```

```
8
6
YES
5
```

1.4.4 Functions without arguments with return type

- These functions will not take any arguments but will return something
- As these functions will have a return type function calls must be made in the same way there were made for previous type of functions
- These functions will be used in abstraction of code most of the times.
- Syntax

```
def function_name(): # Remember no parameters
    process
    return something
```

- Example

```
def get_int():
    return int(input())
```

The above function takes nothing as arguments but returns an integer after reading it from user

[13]:
```
# Assume that you have to read an integer multiple times in the program, but␣
 ↪every time you
# want to read an integer you'd have to write int(input())
a = int(input())
b = int(input())
c = int(input())
d = int(input())
e = int(input())
print(a + b + c + d + e)
```

```
10
20
30
40
50
150
```

[16]:
```python
# Instead of above approach we can define a user function that does the same␣
 ↪with minimum characters
# in the function call

def gi(): # this get_i() function reads an integers and returns it
    return int(input())

# Using the function to read integers as inputs
a = gi()
b = gi()
c = gi()
d = gi()
e = gi()
print(a + b + c + d + e)
```

```
10
20
30
40
50
150
```

[17]:
```python
def gs():
    return input()

def gi():
    return int(gs())

def gf():
    return float(gs())

roll_number = gi()
percentage = gf()
name = gs()

print(f'{name} with {roll_number} has scored {percentage} in the exam')
```

```
12345
76.50
Tokyo
Tokyo with 12345 has scored 76.5 in the exam
```

1.5 Positional, Keyword and Default Arguments in Functions

[23]:
```python
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Mumbai", "India", "Asia") # Positional Arguments
```

```
Mumbai is in India, which is in Asia
```

[24]:
```python
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("India", "Asia", "Mumbai") # Positional Arguments
```

```
India is in Asia, which is in Mumbai
```

[25]:
```python
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location(country="India", continent="Asia", city="Mumbai") # Keyword Arguments
```

```
Mumbai is in India, which is in Asia
```

[27]:
```python
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location(Country="India", continent="Asia", city="Mumbai") # Keyword Arguments
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4888\4099410808.py in <cell line: 6>()
      4
      5 # function call
----> 6 location(Country="India", continent="Asia", city="Mumbai") # Keyword␣
 ↪Arguments

TypeError: location() got an unexpected keyword argument 'Country'
```

[28]:
```python
# function definition
def location(city, country, continent): # Parameters
```

```
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Mumbai", "India", "Asia") #
```

```
Mumbai is in India, which is in Asia
```

[29]:
```
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Mumbai", "India") #
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4888\2185690182.py in <cell line: 6>()
      4
      5 # function call
----> 6 location("Mumbai", "India") #

TypeError: location() missing 1 required positional argument: 'continent'
```

[30]:
```
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Mumbai")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4888\1401860566.py in <cell line: 6>()
      4
      5 # function call
----> 6 location("Mumbai")

TypeError: location() missing 2 required positional arguments: 'country' and␣
 ↪'continent'
```

[31]:
```
# function definition
def location(city, country, continent): # Parameters
    print(f'{city} is in {country}, which is in {continent}')

# function call
```

```
location()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4888\2306759804.py in <cell line: 6>()
      4
      5 # function call
----> 6 location()

TypeError: location() missing 3 required positional arguments: 'city',␣
 ↪'country', and 'continent'
```

[32]:
```
# function definition
def location(city, country, continent="Asia"): # Default Argument
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Mumbai", "India")
```

```
Mumbai is in India, which is in Asia
```

[35]:
```
# function definition
def location(city, country, continent="Asia"): # Default Argument
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("New York", "US", "North America")
```

```
New York is in US, which is in North America
```

[39]:
```
# function definition
def location(city="Mumbai", country="India", continent="Asia"): # Default␣
 ↪Argument
    print(f'{city} is in {country}, which is in {continent}')

# function call
location()
location("Hyderabad")
location("Tokyo", "Japan")
location("Berlin", "Germany", "Europe")
```

```
Mumbai is in India, which is in Asia
Hyderabad is in India, which is in Asia
Tokyo is in Japan, which is in Asia
Berlin is in Germany, which is in Europe
```

1.5.1 No Postional arguments should follow a keyword/default argument argument

[40]:
```
# function definition
def location(city, country="India", continent="Asia"): # Default Argument
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Berlin", "Germany", "Europe")
```

```
Berlin is in Germany, which is in Europe
```

[41]:
```
# function definition
def location(city = "Mumbai", country, continent="Asia"): # Default Argument
    print(f'{city} is in {country}, which is in {continent}')

# function call
location("Berlin", "Germany", "Europe")
```

```
  File "C:\Users\Unstoppable Force\AppData\Local\Temp\ipykernel_4888\657768532.
↪py", line 2
    def location(city = "Mumbai", country, continent="Asia"): # Default Argumen
                                           ^
SyntaxError: non-default argument follows default argument
```

1.6 Arbitrary Number of Arguments to a function or Variable Sized Arguments

[4]:
```
def largest_of_2(a, b):
    if a > b:
        return a
    else:
        return b

def largest_of_3(a: int, b: int, c: int) -> int:
    if a > b and a > c:
        return a
    elif b > a and b > c:
        return b
    else:
        return c


print(largest_of_3(10, 20, 30))
print(largest_of_2(100, 200, 300))
```

```
30
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_30612\3108105450.py in <cell line: 17>()
     15
     16 print(largest_of_3(10, 20, 30))
---> 17 print(largest_of_2(100, 200, 300))

TypeError: largest_of_2() takes 2 positional arguments but 3 were given
```

- From above example we can see that we cannot call a function with more number of arguments than the the ones defined in the definition
- But how come the functions like max, min, math.lcm works with n number of individual arguments?
- And what should we do if we want to create such functions?

[5]:
```
print(max(10, 20))
print(max(10, 20, 30))
print(max(10, 20, 30, 40))
print(max(10, 20, 30, 40, 50))
```

```
20
30
40
50
```

[16]:
```
def multiply(*nums):
    pro = 1
    # nums is a tuple that contains
    # all the elements passed as
    # arguments
    for i in nums:
        pro *= i
    return pro
# Function calls with arbitrary number of arguments
print(multiply(10, 20))
print(multiply(10, 20, 30))
print(multiply(10, 20, 30, 40))
print(multiply(10, 20, 30, 40, 50))
```

[16]:
```
200
```

2 Doc Strings in Functions

- Used to write some documentation about the function we are about to created
- Should be written before writing any function body.
- Should use single or double quotes for single lined Doc String
- Should use triple quotes for multi-line Doc String

[1]:
```
# Here pow() functions doc string is written by the developers of Python
# We can use help() to fetch the doc string of any function (Including User␣
 ↪Defined)
help(pow)
```

```
Help on built-in function pow in module builtins:

pow(base, exp, mod=None)
    Equivalent to base**exp with 2 arguments or base**exp % mod with 3 arguments

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

[41]:
```
def is_prime(n: int) -> bool:
    """
    This function
    Takes: an integer n
    Returns:
    True --> If n is prime
    False --> If n is not prime
    """
    cnt = 0
    for i in range(1, n + 1):
        if n % i == 0:
            cnt += 1
    return cnt == 2
# By writing help(is_prime) we can get the doc string written in that function
help(is_prime)
```

```
Help on function is_prime in module __main__:

is_prime(n: int) -> bool
    This function
    Takes: an integer n
    Returns:
    True --> If n is prime
    False --> If n is not prime
```

[42]:
```
def is_prime(n: int) -> bool:
    "Neekenduku"
    cnt = 0
    for i in range(1, n + 1):
        if n % i == 0:
            cnt += 1
    return cnt == 2
help(is_prime)
```

```
Help on function is_prime in module __main__:

is_prime(n: int) -> bool
    Neekenduku
```