006_Python_while_Loop

March 9, 2023

1 Loops in Python

Loops in Python programming function similar to loops in C, C++, Java or other languages. Python loops are used to repeatedly execute a block of statements until a given condition returns to be `False`. In Python, we have **two types of looping statements**, namely:

2 Python `while` Loop

Loops are used in programming to repeat a specific block of code. In this article, you will learn to create a `while` loop in Python. We use a `while` loop when we want to repeat a code block.

2.1 What is `while` loop in Python?

The `while` loop in Python is used to iterate over a block of code as long as the expression/condition is `True`. When the condition becomes `False`, execution comes out of the loop immediately, and the first statement after the `while` loop is executed.

We generally use this loop when we don't know the number of times to iterate beforehand.

Python interprets any non-zero value as `True`. `None` and `0` are interpreted as `False`.

2.2 Why and When to use `while` loop in Python

Now, the question might arise: when do we use a `while` loop, and why do we use it.

- **Automate and repeat tasks:** As we know, `while` loops execute blocks of code over and over again until the condition is met it allows us to automate and repeat tasks in an efficient manner.
- **Indefinite Iteration:**The `while` loop will run as often as necessary to complete a particular task. When the user doesn't know the number of iterations before execution, `while` loop is used instead of a [**for loop**] loop
- **Reduce complexity:** `while` loop is easy to write. using the loop, we don't need to write the statements again and again. Instead, we can write statements we wanted to execute again and again inside the body of the loop thus, reducing the complexity of the code
- **Infinite loop:** If the code inside the `while` loop doesn't modify the variables being tested in the loop condition, the loop will run forever.

2.2.1 Syntax:

```
while condition:
    body of while loop
```

1. In the **while** loop, expression/condition is checked first.
2. The body of the loop is entered only if the expression/condition evaluates to **True**.
3. After one iteration, the expression/condition is checked again. This process continues until the test_expression evaluates to **False**.

Note: An infinite loop occurs when a program keeps executing within one loop, never leaving it. To exit out of infinite loops on the command line, press **CTRL + C**.

[1]:
```
# Example 1: Print numbers less than 5

count = 1
# run loop till count is less than 5
while count < 5:
    print(count)
    count = count + 1
```

```
1
2
3
4
```

[2]:
```
# Example 2:

num = 10
sum = 0
i = 1
while i <= num:
    sum = sum + i
    i = i + 1
print("Sum of first 10 number is:", sum)
```

```
Sum of first 10 number is: 55
```

[3]:
```
# Example 3:

a=10        # 'a' is my variable

while a>0:  # Enter the body of while loop because condition is TRUE
    print (("Value of a is"),a)
    a=a-2
print ("Loop is Completed")
```

```
Value of a is 10
Value of a is 8
Value of a is 6
```

```
Value of a is 4
Value of a is 2
Loop is Completed
```

[4]:
```
# Example 4:

n=153
sum=0

while n>0:
    r=n%10   # r is the remainder of the division
    sum+=r   # sum+=r is equal to sum = sum+r
    n=n/10
print (sum)
```

```
9.999999999999998
```

[5]:
```
# Example 5: How many times a given number can be divided by 3 before it is␣
 ↪less than or equal to 10.

count = 0
number = 180
while number > 10:
    # divide number by 3
    number = number / 3
    # increase count
    count = count + 1
print('Total iteration required', count)
```

```
Total iteration required 3
```

[6]:
```
# Example 6: Program to add natural numbers up to sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter, i.e., the value of i will change from 1 to 2␣
 ↪in next iteration...
```

```
# print the sum
print("The sum is", sum)
```

```
The sum is 55
```

Explanation:

In the above program, the test expression will be **True** as long as our counter variable **i** is less than or equal to **n** (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop).

Finally, the result is displayed.

[7]:
```
# Example 7: simple fibonacci series
# the sum of two elements defines the next set

a, b = 0, 1
while b < 1000:
    print(b, end = ' ', flush = True)
    a, b = b, a + b

print() # line ending
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

2.3 while loop with if-else

A **while** loop can have an optional [**if-else**] block. We use **if-else** statement in the loop when conditional iteration is needed. i.e., If the condition is **True**, then the statements inside the if block will execute othwerwise, the else block will execute.

[8]:
```
# Example 1: Print even and odd numbers between 1 to the entered number.

n = int(input('Please Enter Number '))
while n > 0:
    # check even and odd
    if n % 2 == 0:
        print(n, 'is a even number')
    else:
        print(n, 'is a odd number')
    # decrease number by 1 in each iteration
    n = n - 1
```

```
Please Enter Number 9
9 is a odd number
8 is a even number
7 is a odd number
6 is a even number
```

```
5 is a odd number
4 is a even number
3 is a odd number
2 is a even number
1 is a odd number
```

[9]:
```python
# Example 2: printing the square of odd numbers less than n.

n =10
i = 1
while i < n:
    #if (i % 2 == 0): (for even numbers)
    if (i % 2 != 0):
        print(i ** 2)
        i = i + 1
    else:
        print(i)
        i = i + 1
```

```
1
2
9
4
25
6
49
8
81
```

[10]:
```python
# Example 3: Add all even numbers from 1 to 10 using while loop
# 2+4+6+8+10

# n = int(input("Please enter the maximum value: "))
n = 10

sum = 0
i = 1

while i <= n:
    if(i%2==0):
        sum = sum + i
#         sum += i
    i = i+1

# print the sum
print("The sum is", sum)
```

```
The sum is 30
```

[11]:
```
# Example 4: Write a code to add all the prime numbers between 17 to 53 using␣
 ↪while loop
# 17, 19, 23, 29, 31, 37, 41, 43, 47, 53

'''Method 1'''

sum=0
for i in range(17,54):
    k=2
    if i>=2:
        while i % k!=0:
            k+=1
        if i==k:
            sum += i
            print(i)
print("The total sum is",sum)
```

```
17
19
23
29
31
37
41
43
47
53
The total sum is 340
```

2.4 `while` loop with `else`

A **`while`** loop can have an optional **`else`** block as well. The **`else`** part is executed if the condition in the **`while`** loop evaluates to **`False`**.

The **`else`** will be skipped/ignored when:

- **`while`** loop terminates abruptly
- The [**break statement**] is used to break the **`while`** loop

[12]:
```
count = 0
while count < 5:
    print(count)
    count = count + 1
else:
    print(count)
```

```
0
1
2
```

```
3
4
5
```

Explanation:

The above loop condition will be false when count is 5 and the loop stops, and execution starts the else statement. As a result 5 will be printed.

[13]:
```
# Example 1: Use while loop to print numbers from 1 to 6

i = 1
while i <= 6:
    print(i)
    i = i + 1
else:
    print("Done. 'while loop' executed normally")
```

```
1
2
3
4
5
6
Done. 'while loop' executed normally
```

[14]:
```
# Example 2: Else block with break statement in a while loop.

i = 1
while i <= 6:
    print(i)
    if i == 3:
        break
    i = i + 1
else:
    print("Done. `while loop` executed normally")
```

```
1
2
3
```

[15]:
```
# Example 3:
'''Example to illustrate the use of else statement with the while loop'''

counter = 0  # counter is my variable

while counter < 3:
    print("Inside while loop")
    counter = counter + 1  # increment the counter
```

```
else:
    print("Inside else")
```

```
Inside while loop
Inside while loop
Inside while loop
Inside else
```

Explanation:

Here, we use a **counter** variable to print the string **Inside loop** three times.

On the fourth iteration, the condition in **while** becomes **False**. Hence, the **else** part is executed.

[16]:
```
# Example 4: we want a user to enter any number between 100 and 600

number = int(input('Enter any number between 100 and 600 '))
# number greater than 100 and less than 600
while number < 100 or number > 600:
    print('Incorrect number, Please enter correct number:')
    number = int(input('Enter a Number between 100 and 600 '))
else:
    print("Given Number is correct", number)
```

```
Enter any number between 100 and 600 555
Given Number is correct 555
```

2.5 Using Control Statement in `while` loops in Python

[**Control statements**] in Python like **break**, **continue**, etc can be used to control the execution flow of **while** loop in Python. Let us now understand how this can be done.

It is used when you want to exit a loop or skip a part of the loop based on the given condition. It also knows as **transfer statements**.

2.5.1 a) `break` in `while` loop

Using the **break** statement, we can exit from the **while** loop even if the condition is **True**.

If the **break** statement is used inside a nested loop (loop inside another loop), it will terminate the innermost loop. For example,

[17]:
```
# Example 1:

count = 0
while count < 5:
    print(count)
    count = count + 1
    if count == 3:
        break
```

```
0
1
2
```

Explanation:

Here, the **while** loop runs until the value of the variable **i** is less than 5. But because of the **break** statement, the loop gets terminated when the value of the variable **i** is 3 and it prints 0, 1, 2

[18]:
```
# Example 2:

list = [60, "HelloWorld", 90.45, 50, 67.23, "Python"]  # total 6 elements
i = 0
while(i < 6):
    print(list[i])
    i = i + 1
    if(i == 3):
        break
```

```
60
HelloWorld
90.45
```

Explanation:

Here, the **while** loop runs until the value of the variable **i** is less than 6. But because of the **break** statement, the loop gets terminated when the value of the variable **i** is 3.

[19]:
```
# Example 3: Display each character from a string and if `a` character is␣
 ↪number then stop the loop.

name = 'Alan99White'
size = len(name)
i = 0
# iterate loop till the last character
while i < size:
    # break loop if current character is number
    if name[i].isdecimal():
        break;
    # print current character
    print(name[i], end=' ')
    i = i + 1
```

```
A l a n
```

2.5.2 b) continue in while loop

The **continue** statement is used to stop/skip the block of code in the loop for the current iteration only and continue with the next iteration.

For example, let's say you want to print all the odd numbers less than a particular value. Here is how you can do it using **continue** keyword in Python.

[20]:
```
# Example 1:

count = 0
while count < 5:
    if count == 3:
        continue
    else:
        print(count)
        count = count + 1
```

```
0
1
2
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-20-7359b7dba965> in <module>
      4 while count < 5:
      5     if count == 3:
----> 6         continue
      7     else:
      8         print(count)

KeyboardInterrupt:
```

Explanation:

The above **while** loop only prints 0, 1, 2 and 4 (skips 3).

[21]:
```
# Example 2: printing odd numbers less than `n`

n=10
i = 1
while (i < n):
    if (i % 2 == 0):
        i = i + 1
        continue  # continue means skip the current loop
    else:
        print (i)
        i = i + 1
```

```
1
3
5
7
```

9

Explanation:

Here, the **continue** statement gets executed when the value of the variable is an even number. This simply means, whenever it is an even number, we simply skip all other statements and execute the next iteration.

[22]:
```
# Example 3: Write a while loop to display only alphabets from a string.

name = 'Alan99White'

size = len(name)
i = -1
# iterate loop till the last character
while i < size - 1:
    i = i + 1
    # skip while loop body if current character is not alphabet
    if not name[i].isalpha():
        continue
    # print current character
    print(name[i], end=' ')
```

```
A l a n W h i t e
```

2.5.3 c) pass in while loop

The **pass** statement is a null statement, i.e., nothing happens when the statement is executed. Primarily it is used in empty functions or classes. When the interpreter finds a pass statement in the program, it returns no operation.

[23]:
```
# Example 1:

n = 4
while n > 0:
    n = n - 1
    pass
```

2.6 Reverse while loop

A reverse loop means an iterating loop in the backward direction. A simple example includes:

- Display numbers from 10 to 1.
- Reverse a string or list

[24]:
```
# Example 1: Reverse a while loop to display numbers from 10 to 1

# reverse while loop
i = 10
while i >= 0:
```

```
    print(i, end=' ')
    i = i - 1
```

```
10 9 8 7 6 5 4 3 2 1 0
```

2.7 Nested `while` loops

Nested while loop is a **`while`** loop inside another **`while`** a loop.

In the nested **`while`** loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop. In each iteration of the outer loop inner loop execute all its iteration.

Syntax:

```
while expression:
    while expression:
        statement(s)
    statement(s)
```

2.7.1 `while` loop inside `while` loop

Example: Nested `while` loop

[25]:
```
# Example: print the first 10 numbers on each line 5 times

i = 1
while i <= 3:
    j = 1
    while j <= 10:
        print(j, end='')
        j = j + 1
    i = i + 1
    print()
```

```
12345678910
12345678910
12345678910
```

Example: Nested `while` loop to print the pattern

```
*
* *
* * *
* * * *
* * * * *
```

[26]:
```
# Example 1: Method 1

i = 1
# outer while loop
```

```
# 5 rows in pattern
while i < 6:
    j = 0
    # nested while loop
    while j < i:
        print('*', end=' ')
        j = j + 1
    # end of nested while loop
    # new line after each row
    print('')
    i = i + 1
```

```
*
* *
* * *
* * * *
* * * * *
```

2.7.2 `for` loop inside `while` loop

We can also use [**for loop**] inside a **`while`** loop as a nested loop. For example,

[27]:
```
# Example 1: Method 2

i = 1
# outer while loop
while i < 6:
    # nested for loop
    for j in range(1, i + 1):
        print("*", end=" ")
    print('')
    i = i + 1
```

```
*
* *
* * *
* * * *
* * * * *
```

[28]:
```
# Example 2: Write a code to add all the prime numbers between 17 to 53 using␣
 ↪while loop
# 17, 19, 23, 29, 31, 37, 41, 43, 47, 53

'''Method 1'''

n=17
sum=0
while n<=53:
```

```
    for i in range(2,n):
        if(n % i)== 0:
            break
        else:
            i=i+1
    else:
        sum=sum+n
        print(n)
    n=n+1
print("The total sum is",sum)
```

```
17
19
23
29
31
37
41
43
47
53
The total sum is 340
```

[29]:
```
# Example 3:

print('Show Perfect number fom 1 to 100')
n = 2

while n <= 100:             # outer while loop
    x_sum = 0
    for i in range(1, n):  # inner for loop
        if n % i == 0:
            x_sum += i
    if x_sum == n:
        print('Perfect number:', n)
    n += 1
```

```
Show Perfect number fom 1 to 100
Perfect number: 6
Perfect number: 28
```

2.8 Iterate String using `while` loop

By looping through the [**string**] using **`while`** loop, we can do lots of string operations. For example,

[30]:
```
# Example 1: while loop to iterate string letter by letter

name = "Alan"
```

```
i = 0
res = len(name) - 1
while i <= res:
    print(name[i])
    i = i + 1
```

```
A
l
a
n
```

2.9 Iterate List using `while` loop

Python list is an ordered collection of items of different data types. It means Lists are ordered by index numbers starting from 0 to the total items-1. List items are enclosed in square `[]` brackets.

Below are the few examples of Python list.

```
>>> numers = [1,2,4,6,7]
>>> players = ["Messi", "Ronaldo", "Neymar"]
```

Using a loop, we can perform various operations on the list. There are ways to iterate through elements in it. Here are some examples to help you understand better.

[31]:
```
# Example 1: Use while loop to iterate over a list.

numbers = [1, 2, 3, 6, 9]
size = len(numbers)
i = 0
while i < size:
    print(numbers[i])
    i = i + 1
```

```
1
2
3
6
9
```

[32]:
```
# Example 2: printing the first two elements of a list

list = [60, "HelloWorld", 90.96]  # list with three elements
i = 0
while(i < 2):
    print (list[i]) #printing the element in the index i
    i = i + 1
```

```
60
HelloWorld
```

Note: When working with the **`while`** loop, it is important to declare the indexing variable **`i`** beforehand and to increment the indexing variable accordingly. Else it will result in an infinite loop.

2.10 Iterate Numbers using `while` loop

Now, consider a program where you want to print out the squares of all the [**numbers**] less a particular number. Let's see how this works with a while statement.

[33]:
```
# Example 1: printing the square of numbers less than `n`

i = 1
while (i <= 10):
    print (i ** 2) #printing the element in the index i
    i = i + 1
```

```
1
4
9
16
25
36
49
64
81
100
```

[34]:
```
# Example 2: Find the cube of number from 1 to 9.

i = 1
while(i<10):
    print(i*i*i)
#   print(i**3)
    i=i+1
```

```
1
8
27
64
125
216
343
512
729
```

[35]:
```
# Example 3:

i = 1
```

```
while i < 3:
    print(i ** 2)
    i = i+1
print('Loop over')
```

```
1
4
Loop over
```

2.11 While loop in Python FAQs

1. A while loop in python is used for what type of iteration? > A while loop is ideal for iteration when the number of iterations is not known. Also, it is ideal to use a while loop when you have a condition that needs to be satisfied.
2. Why does infinite while loop occur in Python? > A loop becomes an infinite loop if the while condition never becomes FALSE. Such a loop that never terminates is called an infinite loop.
3. Is there a do…while loop in Python? >No. There is no do while loop in Python.
4. How do you break a while loop in Python? >Usually, the control shifts out of the while loop when the while condition is no False. Or you can use various control statements like break, continue, etc to break out of the while loop or break of the particular iteration of the while loop.
5. How to write an empty while function in Python? >You can write an empty while function in Python using the pass statements. Here is how you can write it.

```
#empty while statement in python
>>> i = 0
>>> while (i < 10) :
>>>     pass
```

[ ]:

001_Python_Functions

March 9, 2023

1 Python Functions

In this class, you'll learn about functions, what a function is, the syntax, components, and types of functions. Also, you'll learn to create a function in Python.

2 What is a function in Python?

In Python, a **function is a block of organized, reusable (DRY- Don't Repeat Yourself) code with a name** that is used to perform a single, specific task. It can take arguments and returns the value.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it improves efficiency and reduces errors because of the reusability of a code.

2.1 Types of Functions

Python support two types of functions

1. [**Built-in**] function
2. [**User-defined**] function

1.**Built-in function**

The functions which are come along with Python itself are called a built-in function or predefined function. Some of them are: `range()`, `print()`, `input()`, `type()`, `id()`, `eval()` etc.

Example: Python `range()` function generates the immutable sequence of numbers starting from the given start integer to the stop integer.

```
>>> for i in range(1, 10):
>>>     print(i, end=' ')
```

```
1 2 3 4 5 6 7 8 9
```

2. **User-defined function**

Functions which are created by programmer explicitly according to the requirement are called a user-defined function.

Syntax:

```
def function_name(parameter1, parameter2):
    """docstring"""
    # function body
    # write some action
return value
```

2.2 Defining a Function

1. **def** is a keyword that marks the start of the function header.
2. **function_name** to uniquely identify the function. Function naming follows the same [**rules of writing identifiers in Python**].
3. **parameter** is the value passed to the function. They are optional.
4. **:** (colon) to mark the end of the function header.
5. **function body** is a block of code that performs some task and all the statements in **function body** must have the same **indentation** level (usually 4 spaces).
6. **“ “ “docstring” “ “** documentation string is used to describe what the function does.
7. **return** is a keyword to return a value from the function.. A return statement with no arguments is the same as return **None**.

   Note: While defining a function, we use two keywords, **def** (mandatory) and **return** (optional).

Example:

```
>>> def add(num1,num2):            # Function name: 'add', Parameters: 'num1', 'num2'
>>>     print("Number 1: ", num1) #  Function body
>>>     print("Number 2: ", num2) #  Function body
>>>     addition = num1 + num2    #  Function body
>>>     return addition           # return value


>>> res = add(2, 4)    # Function call
>>> print("Result: ", res)
```

2.3 Defining a function without any parameters

Function can be declared without parameters.

[1]:
```
# Example 1:

def greet():
    print("Welcome to Python for Data Science")

# call function using its name
greet()
```

```
Welcome to Python for Data Science
```

[2]:
```
# Example 2:

def add_two_numbers ():
    num_one = 3
    num_two = 6
    total = num_one + num_two
    print(total)
add_two_numbers() # calling a function
```

```
9
```

[3]:
```
# Example 3:

def generate_full_name ():
    first_name = 'Milaan'
    last_name = 'Parmar'
    space = ' '
    full_name = first_name + space + last_name
    print(full_name)
generate_full_name () # calling a function
```

```
Milaan Parmar
```

2.4 Defining a function without parameters and return value

Function can also return values, if a function does not have a **return** statement, the value of the function is None. Let us rewrite the above functions using **return**. From now on, we get a value from a function when we call the function and print it.

[4]:
```
# Example 1:

def add_two_numbers ():
    num_one = 3
    num_two = 6
    total = num_one + num_two
    return total
print(add_two_numbers())
```

```
9
```

[5]:
```
# Example 2:

def generate_full_name ():
    first_name = 'Milaan'
    last_name = 'Parmar'
    space = ' '
    full_name = first_name + space + last_name
    return full_name
```

```
print(generate_full_name())
```

```
Milaan Parmar
```

2.5 Defining a function with parameters

In a function we can pass different data types(number, string, boolean, list, tuple, dictionary or set) as a parameter.

2.5.1 Single Parameter:

If our function takes a parameter we should call our function with an argument

[6]:
```
# Example 1: Gereeting

def greet(name):
    """
    This function greets to the person passed in as a parameter
    """
    print("Hello, " + name + ". Good morning!")    # No output!
```

[7]:
```
# Example 2:

def sum_of_numbers(n):
    total = 0
    for i in range(n+1):
        total+=i
    print(total)
print(sum_of_numbers(10))  # 55
print(sum_of_numbers(100)) # 5050
```

```
55
None
5050
None
```

2.5.2 Two Parameter:

A function may or may not have a parameter or parameters. A function may also have two or more parameters. If our function takes parameters we should call it with arguments.

[8]:
```
# Example 1:

def course(name, course_name):
    print("Hello", name, "Welcome to Python for Data Science")
    print("Your course name is", course_name)

course('Arthur', 'Python')   # call function
```

```
Hello Arthur Welcome to Python for Data Science
Your course name is Python
```

2.6 Defining a function with parameters and return value

[9]:
```python
# Example 1:

def greetings (name):  # single parameter
    message = name + ', welcome to Python for Data Science'
    return message

print(greetings('Milaan'))
```

```
Milaan, welcome to Python for Data Science
```

[10]:
```python
# Example 2:

def add_ten(num):  # single parameter
    ten = 10
    return num + ten
print(add_ten(90))
```

```
100
```

[11]:
```python
# Example 3:

def square_number(x):  # single parameter
    return x * x
print(square_number(3))
```

```
9
```

[12]:
```python
# Example 4:

def area_of_circle (r):  # single parameter
    PI = 3.14
    area = PI * r ** 2
    return area
print(area_of_circle(10))
```

```
314.0
```

[13]:
```python
# Example 5:

def calculator(a, b):  # two parameter
    add = a + b
    return add    # return the addition
```

```
result = calculator(30, 6)   # call function & take return value in variable
print("Addition :", result)   # Output Addition : 36
```

```
Addition : 36
```

[14]:
```
# Example 6:

def generate_full_name (first_name, last_name):  # two parameter
    space = ' '
    full_name = first_name + space + last_name
    return full_name
print('Full Name: ', generate_full_name('Milaan','Parmar'))
```

```
Full Name:  Milaan Parmar
```

[15]:
```
# Example 7:

def sum_two_numbers (num_one, num_two):  # two parameter
    sum = num_one + num_two
    return sum
print('Sum of two numbers: ', sum_two_numbers(1, 9))
```

```
Sum of two numbers:  10
```

[16]:
```
# Example 8:

def calculate_age (current_year, birth_year):  # two parameter
    age = current_year - birth_year
    return age;

print('Age: ', calculate_age(2021, 1819))
```

```
Age:  202
```

[17]:
```
# Example 9:

def weight_of_object (mass, gravity):  # two parameter
    weight = str(mass * gravity)+ ' N' # the value has to be changed to a␣
 ↪string first
    return weight
print('Weight of an object in Newtons: ', weight_of_object(100, 9.81))
```

```
Weight of an object in Newtons:  981.0 N
```

2.7 Function return Statement

In Python, to return value from the function, a **return** statement is used. It returns the value of the expression following the returns keyword.

Syntax:

```
def fun():
    statement-1
    statement-2
    statement-3
    .
    .
    return [expression]
```

The **return** value is nothing but a outcome of function.

- The **return** statement ends the function execution.
- For a function, it is not mandatory to return a value.
- If a **return** statement is used without any expression, then the **None** is returned.
- The **return** statement should be inside of the function block.

2.7.1 Return Single Value

[18]:
```
print(greet("Cory"))
```

```
Hello, Cory. Good morning!
None
```

Here, **None** is the returned value since **greet()** directly prints the name and no **return** statement is used.

2.8 Passing Arguments with Key and Value

If we pass the arguments with key and value, the order of the arguments does not matter.

[19]:
```
# Example 1:

def print_fullname(firstname, lastname):
    space = ' '
    full_name = firstname  + space + lastname
    print(full_name)
print(print_fullname(firstname = 'Milaan', lastname = 'Parmar'))
```

```
Milaan Parmar
None
```

[20]:
```
# Example 2:

def add_two_numbers (num1, num2):
    total = num1 + num2
    print(total)
print(add_two_numbers(num2 = 3, num1 = 2)) # Order does not matter
```

```
5
None
```

If we do not **return** a value with a function, then our function is returning **None** by default. To return a value with a function we use the keyword **return** followed by the variable we are returning. We can return any kind of data types from a function.

[21]:
```
# Example 1:  with return statement

def print_fullname(firstname, lastname):
    space = ' '
    full_name = firstname  + space + lastname
    return full_name
print(print_fullname(firstname = 'Milaan', lastname = 'Parmar'))
```

```
Milaan Parmar
```

[22]:
```
# Example 2:  with return statement

def add_two_numbers (num1, num2):
    total = num1 + num2
    return total
print(add_two_numbers(num2 = 3, num1 = 2)) # Order does not matter
```

```
5
```

[23]:
```
# Example 3:

def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))
print(absolute_value(-4))
```

```
2
4
```

[24]:
```
# Example 4:

def sum(a,b):  # Function 1
    print("Adding the two values")
    print("Printing within Function")
    print(a+b)
    return a+b
```

```
def msg():  # Function 2
    print("Hello")
    return

total=sum(10,20)
print('total : ',total)
msg()
print("Rest of code")
```

```
Adding the two values
Printing within Function
30
total :  30
Hello
Rest of code
```

[25]:
```
# Example 5:

def is_even(list1):
    even_num = []
    for n in list1:
        if n % 2 == 0:
            even_num.append(n)
    # return a list
    return even_num

# Pass list to the function
even_num = is_even([2, 3, 46, 63, 72, 83, 90, 19])
print("Even numbers are:", even_num)
```

```
Even numbers are: [2, 46, 72, 90]
```

2.8.1 Return Multiple Values

You can also return multiple values from a function. Use the return statement by separating each expression by a comma.

[26]:
```
# Example 1:

def arithmetic(num1, num2):
    add = num1 + num2
    sub = num1 - num2
    multiply = num1 * num2
    division = num1 / num2
    # return four values
    return add, sub, multiply, division

a, b, c, d = arithmetic(10, 2)  # read four return values in four variables
```

```
print("Addition: ", a)
print("Subtraction: ", b)
print("Multiplication: ", c)
print("Division: ", d)
```

```
Addition:  12
Subtraction:  8
Multiplication:  20
Division:  5.0
```

2.8.2 Return Boolean Values

[27]:
```
# Example 1:

def is_even (n):
    if n % 2 == 0:
        print('even')
        return True    # return stops further execution of the function, similar to break
    return False
print(is_even(10)) # True
print(is_even(7)) # False
```

```
even
True
False
```

2.8.3 Return a List

[28]:
```
# Example 1:

def find_even_numbers(n):
    evens = []
    for i in range(n + 1):
        if i % 2 == 0:
            evens.append(i)
    return evens
print(find_even_numbers(10))
```

```
[0, 2, 4, 6, 8, 10]
```

2.9 How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

[29]:
```
greet('Alan')
```

```
Hello, Alan. Good morning!
```

Note: Try running the above code in the Python program with the function definition to see the output.

[30]:
```python
# Example 1:

def wish(name):
    """
    This function wishes to the person passed in as a parameter
    """
    print("Happy birthday, " + name + ". Hope you have a wonderful day!")

wish('Bill')
```

```
Happy birthday, Bill. Hope you have a wonderful day!
```

[31]:
```python
# Example 2:

def greetings (name = 'Clark'):
    message = name + ', welcome to Python for Data Science'
    return message
print(greetings())
print(greetings('Milaan'))
```

```
Clark, welcome to Python for Data Science
Milaan, welcome to Python for Data Science
```

[32]:
```python
# Example 3:

def generate_full_name (first_name = 'Milaan', last_name = 'Parmar'):
    space = ' '
    full_name = first_name + space + last_name
    return full_name
print(generate_full_name())
print(generate_full_name('Ethan','Hunt'))
```

```
Milaan Parmar
Ethan Hunt
```

[33]:
```python
# Example 4:

def calculate_age (birth_year,current_year = 2021):
    age = current_year - birth_year
    return age;
print('Age: ', calculate_age(1821))
```

```
Age:  200
```

[34]:
```
# Example 5:

def swap(x, y):
    """
    This function swaps the value of two variables
    """
    temp = x;  # value of x will go inside temp
    x = y;     # value of y will go inside x
    y = temp;  # value of temp will go inside y
    print("value of x is:", x)
    print("value of y is:", y)
    return     # "return" is optional

x = 6
y = 9
swap(x, y)     #call function
```

```
value of x is: 9
value of y is: 6
```

[35]:
```
# Example 6:

def even_odd(n):
    if n % 2 == 0:   # check number is even or odd
        print(n, 'is a Even number')
    else:
        print(n, 'is a Odd Number')

even_odd(9)   # calling function by its name
```

```
9 is a Odd Number
```

[36]:
```
# Example 7:

def weight_of_object (mass, gravity = 9.81):
    weight = str(mass * gravity)+ ' N' # the value has to be changed to string␣
 ↪first
    return weight
print('Weight of an object in Newtons: ', weight_of_object(100)) # 9.81 -␣
 ↪average gravity on Earth's surface
print('Weight of an object in Newtons: ', weight_of_object(100, 1.62)) #␣
 ↪gravity on the surface of the Moon
```

```
Weight of an object in Newtons:  981.0 N
Weight of an object in Newtons:  162.0 N
```

2.10 Docstrings

The first string after the function header is called the **docstring** and is short for documentation string. It is a descriptive text (like a comment) written by a programmer to let others know what block of code does.

Although **optional**, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

It is being declared using triple single quotes `''' '''` or triple-double quote `""" """` so that docstring can extend up to multiple lines.

We can access docstring using doc attribute **`__doc__`** for any object like list, tuple, dict, and user-defined function, etc.

In the above example, we have a docstring immediately below the function header.

[37]:
```
print(greet.__doc__)
```

```
    This function greets to the person passed in as a parameter
```

To learn more about docstrings in Python, visit [**Python Docstrings**].

2.11 Function `pass` Statement

In Python, the **`pass`** is the keyword, which won't do anything. Sometimes there is a situation where we need to define a syntactically empty block. We can define that block using the **`pass`** keyword.

When the interpreter finds a **`pass`** statement in the program, it returns no operation.

[38]:
```
# Example 1:

def addition(num1, num2):
    # Implementation of addition function in comming release
    # Pass statement
    pass

addition(10, 2)
```

2.12 Exercises Functions

2.12.1 Exercises Level 1

1. Area of a circle is calculated as follows: **area = x r x r** and **perimeter = 2 x x r**. Write a function that calculates **`area_of_circle`** and **`perimeter_of_circle`**.
2. Write a function called **`add_all_nums`** which takes arbitrary number of arguments and sums all the arguments. Check if all the list items are number types. If not do give a reasonable feedback.

3. Temperature in **°C** can be converted to **°F** using this formula: **°F = (°C x 9/5) + 32**. Write a function which converts **°C to °F**, **convert_celsius_2_fahrenheit**.
4. Write a function called **check_season**, it takes a month parameter and returns the season: Autumn, Winter, Spring or Summer.
5. Write a function called **calculate_slope** which return the slope of a linear equation
6. Quadratic equation is calculated as follows: **$ax^2 + bx + c = 0$**. Write a function which calculates solution set of a quadratic equation, **solve_quadratic_eqn**.
7. Declare a function named **print_list**. It takes a list as a parameter and it prints out each element of the list.
8. Declare a function named **reverse_list**. It takes an array as a parameter and it returns the reverse of the array (use loops).
   - ```
     print(reverse_list([1, 2, 3, 4, 5]))
     #[5, 4, 3, 2, 1]
     print(reverse_list1(["A", "B", "C"]))
     #["C", "B", "A"]
     ```
9. Declare a function named **capitalize_list_items**. It takes a list as a parameter and it returns a capitalized list of items
10. Declare a function named **add_item**. It takes a list and an item parameters. It returns a list with the item added at the end.
    - ```
      food_staff = ['Potato', 'Tomato', 'Mango', 'Milk']
      print(add_item(food_staff, 'Fungi'))  #['Potato', 'Tomato', 'Mango', 'Milk', 'Fungi']
      numbers = [2, 3, 7, 9]
      print(add_item(numbers, 5))       #[2, 3, 7, 9, 5]
      ```
11. Declare a function named **remove_item**. It takes a list and an item parameters. It returns a list with the item removed from it.
    - ```
      food_staff = ['Potato', 'Tomato', 'Mango', 'Milk']
      print(remove_item(food_staff, 'Mango'))  # ['Potato', 'Tomato', 'Milk']
      numbers = [2, 3, 7, 9]
      print(remove_item(numbers, 3))  # [2, 7, 9]
      ```
12. Declare a function named **sum_of_numbers**. It takes a number parameter and it adds all the numbers in that range.
    - ```
      print(sum_of_numbers(5))  # 15
      print(sum_all_numbers(10)) # 55
      print(sum_all_numbers(100)) # 5050
      ```
13. Declare a function named **sum_of_odds**. It takes a number parameter and it adds all the odd numbers in that range.
14. Declare a function named **sum_of_even**. It takes a number parameter and it adds all the even numbers in that - range.

2.12.2 Exercises Level 2

1. Declare a function named **evens_and_odds**. It takes a positive integer as parameter and it counts number of evens and odds in the number.
    - ```
      print(evens_and_odds(100))
      #The number of odds are 50.
      #The number of evens are 51.
      ```
2. Call your function **factorial**, it takes a whole number as a parameter and it return a factorial of the number
3. Call your function **is_empty**, it takes a parameter and it checks if it is empty or not
4. Write different functions which take lists. They should **calculate_mean**, **calculate_median**, **calculate_mode**, **calculate_range**, **calculate_variance**, **calculate_std** (standard deviation).

2.12.3 Exercises Level 3

1. Write a function called **is_prime**, which checks if a number is prime.
2. Write a functions which checks if all items are unique in the list.
3. Write a function which checks if all the items of the list are of the same data type.
4. Write a function which check if provided variable is a valid python variable
5. Go to the data folder and access the [**countries-data.py**] file.

- Create a function called the **most_spoken_languages** in the world. It should return 10 or 20 most spoken languages in the world in descending order
- Create a function called the **most_populated_countries**. It should return 10 or 20 most populated countries in descending order.

```
[ ]:
```
```

006_Python_Function_Anonymous

March 9, 2023

1 Python Anonymous/`lambda` Function

In this class, you'll learn about the anonymous function, also known as **`lambda`** functions. You'll learn what they are, their syntax and how to use them (with examples).

1.1 What are `lambda` functions in Python?

In Python, an anonymous function is a [**function**] that is defined without a name.

While normal functions are defined using the **`def`** keyword in Python, anonymous functions are defined using the **`lambda`** keyword.

In opposite to a normal function, a Python **`lambda`** function is a single expression. But, in a lambda body, we can expand with expressions over multiple lines using parentheses `()` or a multiline string `""" """`.

For example: **`lambda n:n+n`**

The reason behind the using anonymous function is for instant use, that is, one-time usage and the code is very concise so that there is more readability in the code.

Hence, anonymous functions are also called **`lambda`** functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because **lambda** requires an expression.
- **`lambda`** functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

Syntax:

```
lambda argument_list: expression
```

Let's see an example to print even numbers without a **`lambda`** function and with a **`lambda`** function. See the difference in line of code as well as readability of code.

[1]:
```
# Example 1: Program for even numbers without lambda function

def even_numbers(nums):
    even_list = []
    for n in nums:
        if n % 2 == 0:
            even_list.append(n)
    return even_list

num_list = [10, 9, 16, 78, 2, 3, 7, 1]
ans = even_numbers(num_list)
print("Even numbers are:", ans)
```

```
Even numbers are: [10, 16, 78, 2]
```

[2]:
```
# Example 1: Program for even number with a lambda function

l = [10, 9, 16, 78, 2, 3, 7, 1]
even_nos = list(filter(lambda x: x % 2 == 0, l))
print("Even numbers are: ", even_nos)
```

```
Even numbers are:  [10, 16, 78, 2]
```

[3]:
```
# Example 2: Program to show the use of lambda functions

double = lambda x: x * 2

print(double(6))
```

```
12
```

Explanation:

In the above program, lambda **`x: x * 2`** is the lambda function. Here **`x`** is the argument and **`x * 2`** is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier **`double`**. We can now call it as a normal function. The statement

```
>>> double = lambda x: x * 2
```

is nearly the same as:

```
>>> def double(x):
>>>     return x * 2
```

[4]:
```
# Example 3: Normal Function definition is here

def square(x):
    return x*x
```

```
# anonymous function
sqr = lambda x: x*x

#Calling square function
print("Square of number is",square(10)) #call normal function
print("Square of number is",sqr(2)) #call anonymous function
```

```
Square of number is 100
Square of number is 4
```

1.2 Use of `lambda` Function in python

We use **`lambda`** function when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as **arguments**). **`lambda`** function are used along with built-in functions like **`filter()`**, **`map()`**, **`reduce()`** etc.

1.2.1 `lambda` function use with `filter()`

The **`filter()`** function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to **`True`**.

Here is an example use of **`filter()`** function to filter out only even numbers from a list.

[5]:
```
# Example 1: Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]  # total 8 elements

new_list = list(filter(lambda x: (x%2 == 0), my_list)) # returns the output in␣
 ↪form of a list

print("Even numbers are: ", new_list)
```

```
Even numbers are:  [4, 6, 8, 12]
```

[6]:
```
# Example 2:

l = [-10, 5, 12, -78, 6, -1, -7, 9]
positive_nos = list(filter(lambda x: x > 0, l))
print("Positive numbers are: ", positive_nos)
```

```
Positive numbers are:  [5, 12, 6, 9]
```

1.2.2 `lambda` function with `map()`

The **`map()`** function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of **`map()`** function to double all the items in a list.

[7]:
```
# Example 1: Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]  # total 8 elements
new_list = list(map(lambda x: x * 2, my_list)) # returns the output in form of␣
 ↪a list
print("Double values are: ", new_list)
```

```
Double values are:  [2, 10, 8, 12, 16, 22, 6, 24]
```

[8]:
```
# Example 2:

list1 = [2, 3, 4, 8, 9]
list2 = list(map(lambda x: x*x*x, list1))
print("Cube values are:", list2)
```

```
Cube values are: [8, 27, 64, 512, 729]
```

1.2.3 `lambda` function with `reduce()`

The **`reduce()`** function is used to minimize sequence elements into a single value by applying the specified condition.

The **`reduce()`** function is present in the **`functools`** module; hence, we need to import it using the import statement before using it.

[9]:
```
# Example 1:

from functools import reduce
list1 = [20, 13, 4, 8, 9]
add = reduce(lambda x, y: x+y, list1)
print("Addition of all list elements is : ", add)
```

```
Addition of all list elements is :  54
```

[ ]:
```

```

005_Python_Function_Recursion

March 9, 2023

1 Python Recursion

In this class, you will learn to create a recursive function (a function that calls itself again and again).

2 What is recursion?

Recursion is the process of defining something in terms of itself.

A physical world example would be to **place two parallel mirrors facing each other**. Any object in between them would be reflected recursively.

2.1 Python Recursive Function

In Python, we know that a [**function**] can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called **`recurse`**.

Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is

`1*2*3*4*5*6 = 720.`

[1]:
```
# Example 1:

def factorial(n):
    """This is a recursive function to find the factorial of an integer"""

    if n == 1:
        return 1
    else:
        return (n * factorial(n-1))  # 3 * 2 * 1 = 6


num = 3
print("The factorial of", num, "is", factorial(num))
```

```
The factorial of 3 is 6
```

In the above example, **`factorial()`** is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Let's look at an image that shows a step-by-step process of what is going on:

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is **`1000`**. If the limit is crossed, it results in **`RecursionError`**. Let's look at one such condition.

[2]:
```
def recursor():
    recursor()
recursor()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-2-3ec8c6652eef> in <module>
      1 def recursor():
      2     recursor()
----> 3 recursor()

<ipython-input-2-3ec8c6652eef> in recursor()
      1 def recursor():
----> 2     recursor()
      3 recursor()

… last 1 frames repeated, from the frame below …

<ipython-input-2-3ec8c6652eef> in recursor()
      1 def recursor():
----> 2     recursor()
      3 recursor()
```

```
RecursionError: maximum recursion depth exceeded
```

2.2 Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

2.3 Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

[ ]: