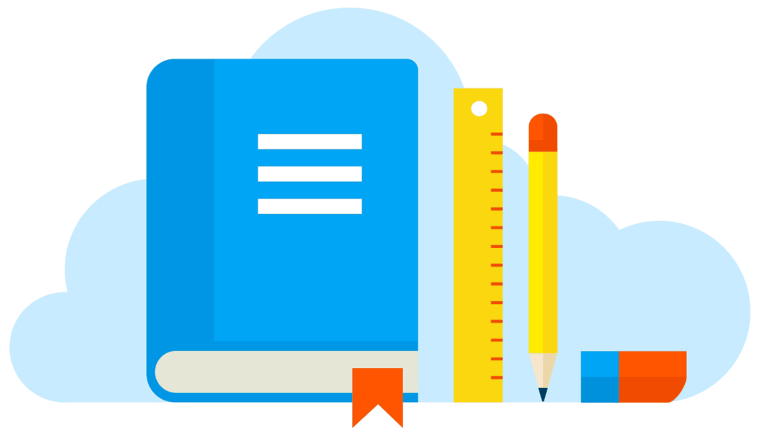


Python Cheatsheet for Quick Reference

WEEK 1

1. `print()` : method to print on the console.

```
print("Hello World")
```

2. Variables are used to store values. The values can be number, text etc.

3.
```
a = 10          #int
n = 10.3        #float
s = "Student"   #str
t = True        #bool
```

4. `input()` method is used to take input from user in the form of string
5. `type()` method is used to get datatype of a variable or value

6. Typecasting:to change a datatype from one to another.

```
a = int(10.1)      #This will convert 10.1 to 10
```

```
a1 = int(10.1)    #This will convert 10.1 to 10
b = float(10)      #This will convert 10 to 10.0
c = str(10.3)      #This will convert 10.3 to ‘10.3’
d = bool(0)        #False
e = bool(10)       #True
f = bool(-10)      #True
g = bool(‘India’)  #True
i = bool(‘’)       #False
```

7. Arithmetic operators

```
+  -> Addition (int or float) , Concatenation (str)
-  -> Subtraction
*  -> Multiplication (int or float) , Repetition (str)
/  -> Division
// -> Floor Division (Integer division)
%  -> Modulus operator (remainder after division)
** -> Exponentiation (power)
```

8.Relational operators(It will compare and give True or False)

```
>  -> Greater than
<  -> Less than
>= -> Greater than or equal
<= -> Less than or equal
== -> Equal
!= -> Not equal
```

9. Logical operators

```
and -> Returns True if both LHS and RHS are true
or  -> Returns False if both LHS and RHS are false
not -> Returns negation of given operand
```

10. Strings

```
s1 = 'Hello'
```

Or

```
s2 = "World"
```

11. String indexing (starts from 0)

```
s1[0] -> H
s1[-1] -> o
```

12. String slicing

```
s1[0:3] -> Hel
```

13. String comparison - (order in English dictionary)

```
'apple' > 'one'  -> False
'apple' > 'ant'  -> True
```

14. String length

len() method gives length of a string.

```
len(s1)  -> 5
```

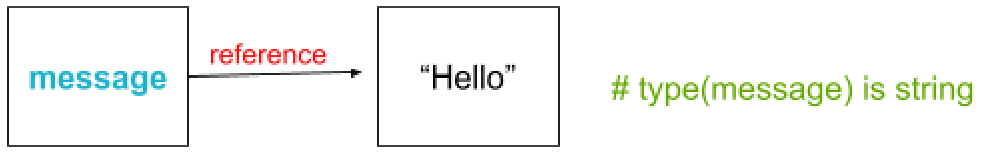
WEEK 2

1. **Comments:** A section of our program that is ignored by the compiler(to increase the readability of the code).

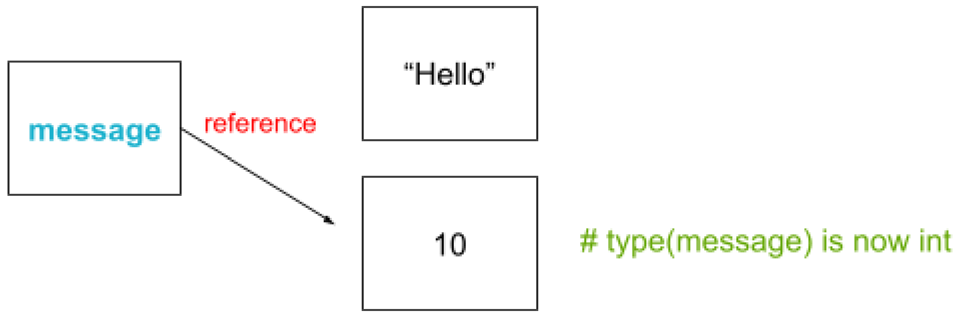
```
-> # This is a comment

-> ''' This is

   A multiline

   Comment '''
```

2. **Dynamic Typing:** The type of a variable in Python is determined by the value it stores and changes with it.

```
-> message = 'Hello'
```



```
-> message = 10
```



3. **Multiple Assignments**

Assign multiple values or the same value to multiple variables

x, y, z= 1 , "Hi" , **False**	x=y=z=100
print(x,y,z) -> 1 "Hi" **False**	print(x,y,z) -> 100 100 100

4. **Del** x -> Delete the variable x from memory

5. **Shorthand Operators**

   += , -= , *= , /= ....etc , Basically any operator with the "=" sign.

   num = num + 1 is same as num +=1

   num = num * 2 is same as num *=2

6. **in** operator - Tells whether something is inside/part of the other thing (similar to the English definition)

   " IT " in " IIT Madras " -> True # searches for the string "IT" in "IIT Madras"

   " ads " in " IIT Madras " -> False

7. **Chaining Operators** - Using multiple relational operators (<,>,==,<=,>=,!=) together

   ```
   x = 5
   print( 1 < x < 6 ) -> True
   print( x < 2 * x < 3 *x > 4 *x ) -> False
       # above can be looked as
       print( 5 < 10 < 15 > 20 ) -> False   # 15 is not greater than 20
   ```

8. **Escape Characters** - To insert characters that are illegal in a string, use an escape character.
   An escape character is a backslash \ followed by the character you want to insert.

   print('It's Raining') -> **Error #All Characters after second ' (single quote) are not considered as a part of the string (s Raining)**

   # So we use \ before the ' print('It\'s Raining') -> It's Raining

   Other Escape Characters - \n : Cursor moves to the next line
   \t: Shifts cursor by 5 spaces

9. **Multiline Strings:** Similar to multiline comment

```
S = ' ' ' A multiline
          String ' ' '

print(S)  -> A multiline
             String
```

10. In Python, variable names must start with a letter (a-z, A-Z) or an underscore (_), and can be followed by letters, digits (0-9), or underscores. They are case-sensitive and cannot use Python-reserved keywords. Variable name cannot start with any digit (0-9)

11. **String Methods**

Method	Description	Code x = 'pytHoN sTrIng mEthOdS'	Output
lower()	Converts a string into lower case	print(x.lower())	python string methods
upper()	Converts a string into upper case	print(x.upper())	PYTHON STRING METHODS
capitalize()	Converts the first character to upper case	print(x.capitalize())	Python string methods
title()	Converts the first character of each word to upper case	print(x.title())	Python String Methods
swapcase()	Swaps cases, lower case becomes upper case and vice versa	print(x.swapcase())	PYThOn StRiNG MeTHoDs

Method	Description	Code	Output

islower()	Returns True if all characters in the string are lower case	x = 'python'
		print(x.islower())
			True

		x = 'Python'
		print(x.islower())
			False

isupper()	Returns True if all characters in the string are upper case	x = 'PYTHON'
		print(x.isupper())
			True

		x = 'PYTHoN'
		print(x.isupper())
			False

istitle()	Returns True if the string follows the rules of a title	x = 'Pyhton String Methods'
		print(x.istitle())
			True

		x = 'Pyhton string methods'
		print(x.istitle())
			False

Method	Description	Code	Output

isdigit()	Returns True if all characters in the string are digits	x = '123'
		print(x.isdigit())
			True

		x = '123abc'
		print(x.isdigit())
			False

isalpha()	Returns True if all characters in the string are in alphabets	x = 'abc'
		print(x.isalpha())
			True

		x = 'abc123'
		print(x.isalpha())
			False

isalnum()	Returns True if all characters in the string are alpha-numeric	x = 'abc123'
		print(x.isalnum())
			True

		x = 'abc123@*#'
		print(x.isalnum())
			False

✓

Method	Description	Code x = '-----Python-----'	Output
strip()	Returns a trimmed version of the string	print(x.strip('-'))	Python
lstrip()	Returns a left trim version of the string	print(x.lstrip('-'))	Python-----
rstrip()	Returns a right trim version of the string	print(x.rstrip('-'))	-----Python

Method	Description	Code x = 'Python'	Output
startswith()	Returns True if the string starts with the specified value	print(x.startswith('P'))	True
		print(x.startswith('p'))	False
endswith()	Returns True if the string ends with the specified value	print(x.endswith('n'))	True
		print(x.endswith('N'))	False

✓

Method	Description	Code x = 'Python String Methods'	Output
count()	Returns the number of times a specified value occurs in a string	print(x.count('t'))	3
		print(x.count('s'))	1
index()	Searches the string for a specified value and returns the position of where it was found	print(x.index('t'))	2
		print(x.index('s'))	20

replace()	Returns a string where a specified value is replaced with a specified value	x = x.replace('S', 's')
		x = x.replace('M', 'm')
		print(x)
			Python string methods

✓

12. **If - elif - else** Conditions:

Syntax:

```
if (condition):
    Statement 1      # indentation is used to show that the Statement 1 is inside the if statement, If the condition is True, Statement 1 is executed
elif (condition):
    Statement 2      # If the previous condition is False but this condition is true, Statement 2 is executed, elif block is optional
elif (condition):
    Statement 3   # If the previous two conditions are False but this condition is true, Statement 3 is executed. There may be an unlimited number of elif blocks.
else:
      Statement      # If all the previous conditions are False, the statement under the else block is executed. Else block is optional
```

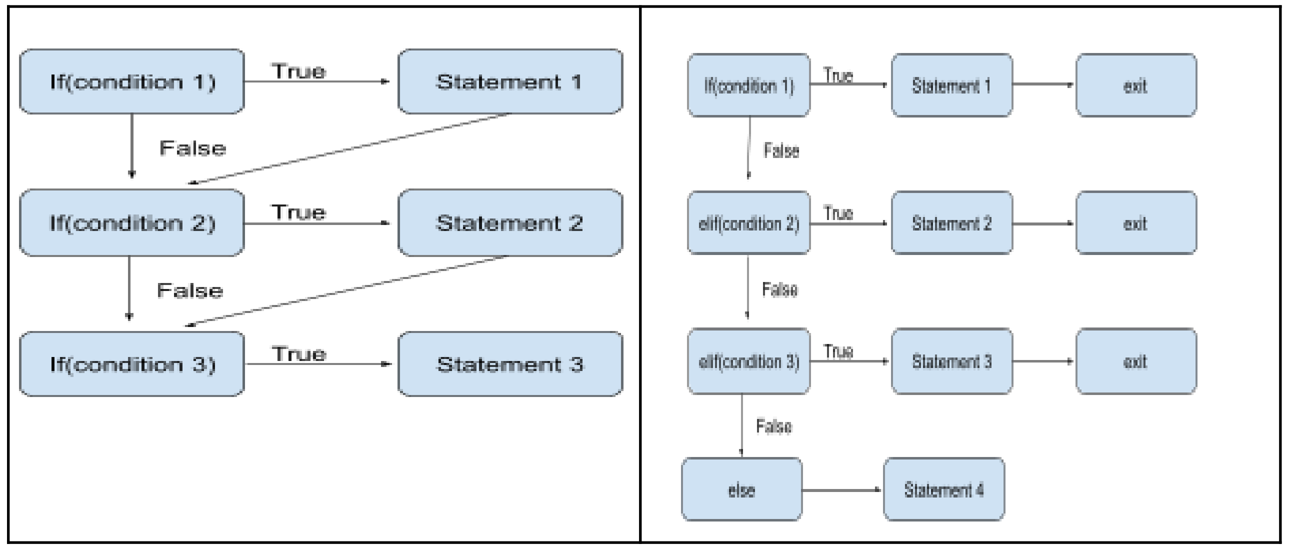
Example 1:

```
if(3<5):
    print(“3 is less than 5”) -> This print statement is executed because the condition inside if is True.
```

Example 2:

```
if(3>5):
     print(“3 is less than 5”)
elif(5>3):
     print(“5 is greater than 3”)     # 'elif` has the same syntax as `if`, can follow an `if` any number of times, and runs only if its condition is true and all previous conditions are false.
else:
     print(“Equal!!!”)                    # else doesn't have a condition and will execute if all the above statements are false
```

Flowchart:



13. **Import library**

1. **import math -> Imports the entire math library and accessing it needs "math" keyword, eg math.sqrt(2)**
2. **from math import * -> Imports the entire math library but "math" keyword is not needed eg. sqrt(2) will work.**

   "*" represents "all" , Instead, we can give one or more function names.

3. **from math import sqrt -> Import only the "sqrt" function from the entire math library**
4. **from math import sqrt as s -> Now sqrt function is stored in the variable named "s"**

   s(2) will now give sqrt(2)

WEEK 3

■ while loop

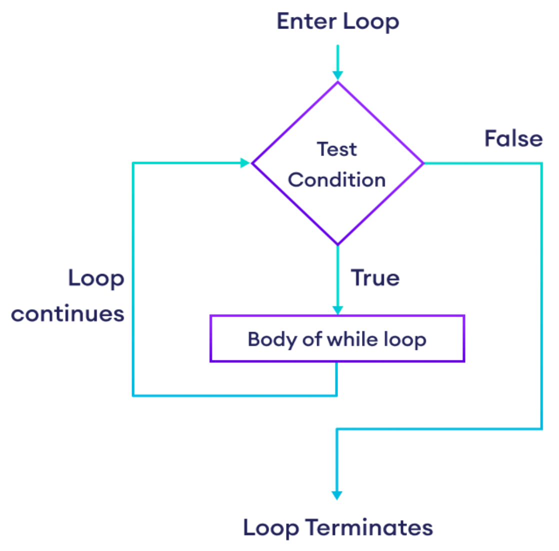
Python While Loop executes a block of statements repeatedly until a given condition is satisfied.

Syntax:

```
while condition:
    # body of while loop
    Statement (s)
```

Note: The condition written beside the while keyword can be placed within round brackets. It is optional in Python.

Flowchart:



Example: Find the factorial of a number

```
n = int(input())    # the number of which factorial we want to find
if (n < 0):
    print("Invalid Input")
else:
    i = 1                # This initialises a counter variable i to 1
    factorial = 1        # This will store the result of the factorial
    while (i <= n):      # the loop runs if the condition is true
        factorial = factorial * i    # logic of factorial
        i += 1           # increment the counter

    print("factorial of", n, "is:", factorial)    # print the answer
```

Output:
```
5
        factorial of 5 is: 120
```

The factorial of a number n (denoted as n!) is the product of all positive integers less than or equal to n. For example, the factorial of 5 (5!) is 5×4×3×2×1=120.

This code uses a while loop to determine the factorial of a given integer by the user. If the number is not positive it shows "Invalid Input", otherwise, it calculates the factorial. The counter, '**i**', initialises to 1 and the '**result**' variable factorial to 1. The while loop continues as long as 'i' is less than or equal to 'n'. Inside the loop, the factorial is updated by multiplying it with '**i**', and then '**i**' is incremented by 1. This process repeats until '**i**' exceeds '**n**', at which point the loop exits. The computed factorial is printed by the code at the end.

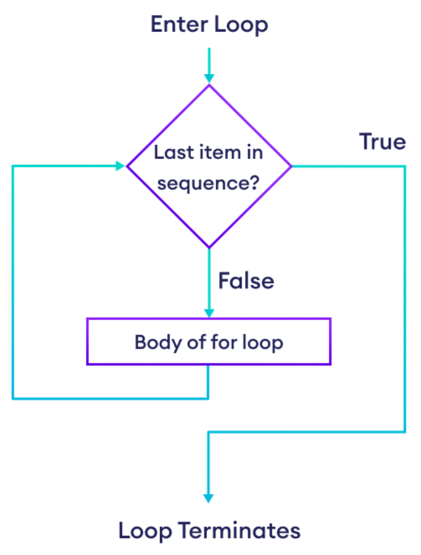
■ for each loop

In Python, a for loop is used to iterate over sequences (that is either a string, a list, a tuple, a dictionary, a set, or a string) or any iterable object.

Syntax:

```
for var in iterable:
    # statements
```

Flowchart:



Let's assume to iterate over a string and print each character on a new line.

```
s = "IITM"
for char in s:
   print(char)
```

Output:

```
I
I
T
M
```

This code iterates over each character in the string s, which is "IITM". The for loop assigns each character to the variable char one by one. Inside the loop, print(char) is called, which prints the current character stored in char.

■ for loop with range()

General Syntax:

`for var in range(start, stop, step)`

By default, the 'start' is 0 and the 'step' is 1

In this case, the range() function creates a series of numbers that starts inclusively from the 'start', increasing by the 'step' up to the 'stop' (non-inclusive). And for each number in that series, the variable, 'var', iterates.

Other syntaxes:

`for var in range(start, stop)`

Here, the range() function creates a series of numbers based on the 'start' (inclusive) and 'stop' (non-inclusive) values that increment by 1.

`for var in range(stop)`

The range() function yields a series of numbers in this case, beginning at 0 and increasing by 1, finishing at the non-inclusive value you specify as 'stop'.

Example:

```
for i in range(3, 11, 2):
    print(i, end = " ")
```

Output: 3 5 7 9

```
for i in range(3, 11):
    print(i, end = " ")
```

Output: 3 4 5 6 7 8 9 10

```
for i in range(11):
    print(i, end = " ")
```

Output: 0 1 2 3 4 5 6 7 8 9 10

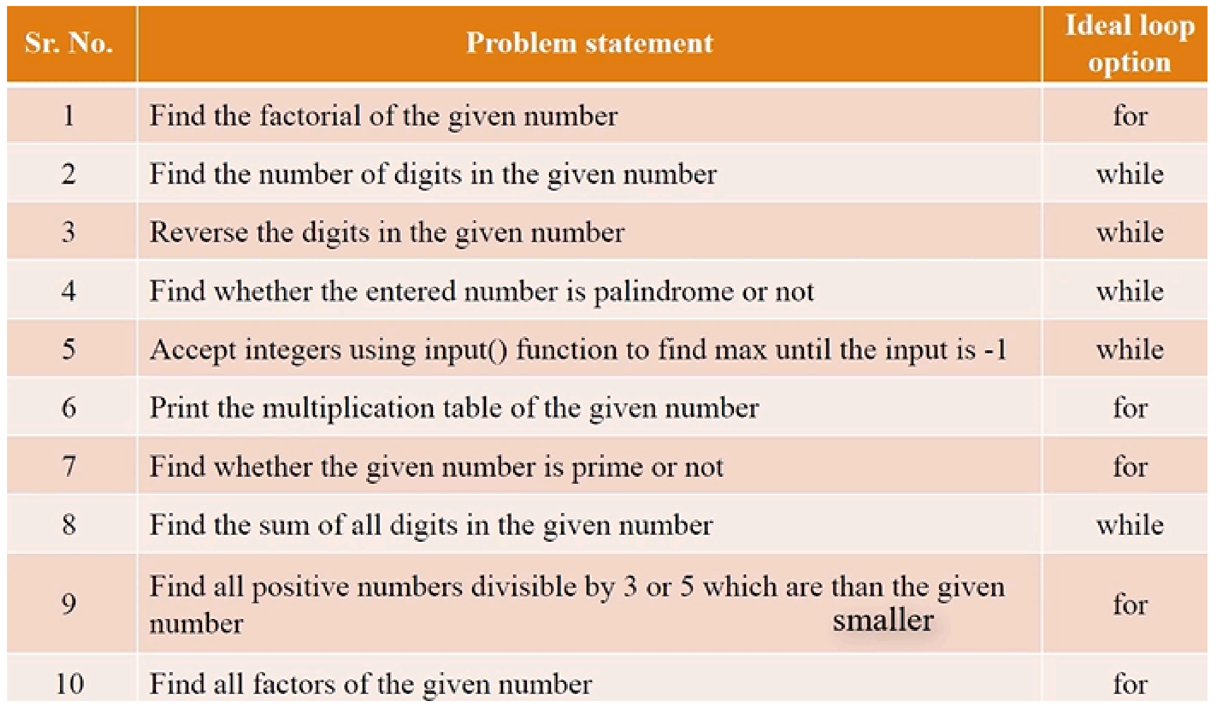
Example: Find the factorial of a number

```
n = int(input())
if (n < 0):
    print("Invalid Input")
else:
    factorial = 1
    for i in range(1, n+1):
        factorial = factorial * i

    print("factorial of", n, "is:", factorial)
```

```
Output: 5
        factorial of 5 is: 120
```

This code calculates the factorial of a non-negative integer n entered by the user. First, it checks if 'n' is less than 0; if so, it prints "Invalid Input" because the factorial is not defined for negative numbers. If 'n' is non-negative, it initializes the variable 'factorial' to 1. The for loop [for i in range(1, n+1):] then iterates from 1 to 'n' (inclusive). In each iteration, the loop multiplies 'factorial' by the current value of 'i', progressively calculating the factorial. Finally, the code prints the result.

■ Ideal loop option based on problem statement



Sr. No.	Problem statement	Ideal loop option
1	Find the factorial of the given number	for
2	Find the number of digits in the given number	while
3	Reverse the digits in the given number	while
4	Find whether the entered number is palindrome or not	while
5	Accept integers using input() function to find max until the input is -1	while
6	Print the multiplication table of the given number	for
7	Find whether the given number is prime or not	for
8	Find the sum of all digits in the given number	while
9	Find all positive numbers divisible by 3 or 5 which are than the given number smaller	for
10	Find all factors of the given number	for

■ Basic difference between for loop and while loop

For Loop	While Loop
It is used when the number of iterations is known.	It is used when the number of iterations is not known.
It has a built-in loop control variable.	There is no built-in loop control variable.

■ Nested loop

Nested loops mean loops inside a loop. For example, while loop inside the for loop, for loop inside the for loop, etc.

Example: Possible unique pair in a string

```
s = "python"
length = len(s)
for i in range(length):
    for j in range(i+1, length):
        print(s[i], s[j])
```

Output:

```
p y
p t
p h
p o
p n
y t
y h
y o
y n
t h
t o
t n
h o
h n
o n
```

This code prints all possible pairs of characters from the string s, which is "python". First, it calculates the length of the string and stores it in the variable length. The outer for loop iterates over each character's index i in the string. For each iteration of the outer loop, the inner for loop iterates over the indices j that are greater than i. Inside the inner loop, it prints the character at index i paired with the character at index j.

- **break:** The break statement, a loop control statement, in Python, terminates the current loop in which it is present..

Example: print a given string until a particular character appears

```python
s = "playing with python"
for char in s:
    if char == 'i':
        break
    print(char, end="")
```

Syntax:

```
for / while loop:
    # statement(s)
    if condition:
        break
    # statement(s)
```

Output: play

The for loop goes through each character in the string, and an if statement checks if the current character is 'i'. If it is, the break statement exits the loop immediately, stopping any further printing. If the character is not 'i', it is printed.

- **continue:** Continue is also a loop control statement that forces to execute the next iteration of the loop skipping the remaining statements of the loop.

Example: print a given string except for a particular character

```python
s = "seeing"
for char in s:
    if char == 'e':
        continue
    print(char, end="")
```

Syntax:

```
for / while loop:
    # statement(s)
    if condition:
        continue
    # statement(s)
```

Output: sing

The for loop goes through each character in the string, and an if statement checks if the current character is 'e'. If it is, the continue statement skips the rest of the loop immediately. If the character is not 'e', it is printed.

- **`pass:`** pass statement simply does nothing. The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. Pass statements can also be used for writing empty loops. Pass is also used for empty control statements, functions, and classes.

Example:

```
s = "seeing"
for char in s:
    if char == 'e':
        pass
    print(char, end="")
```

Syntax:

```
function/condition/loop:
    pass
```

Output: seeing

When the loop encounters the character 'e', the pass statement is executed, which does nothing and allows the loop to continue to the next character. The print statement prints each character on the same line without adding a new line. As a result, the output is "seeing", including the 'e' characters, because pass does not alter the flow of the loop.

Infinite Loop

An infinite loop in Python is a loop that continues to execute indefinitely, or until it is explicitly stopped. Sometimes it occurs unintentionally due to improper updating loop counter.

Example:

```
n, i = 10, 1
while (i <= n):
    print(i)
```

In the above example, we want to print the first 5 whole numbers but forget to increment the loop counter inside the loop. It will print an unlimited number of 1

Done

■ Formatted Printing

❖ **sep parameter in print():** The separator between the arguments to print() function in Python is space by default which can be modified and made to any character using the 'sep' parameter.

```
print("11", "06", "24")                  Output:  11 06 24

print("11", "06", "24", sep = "/")       Output:  11/06/24

print("11", "06", "24", sep = "S")       Output:  11S06S24
```

❖ **end parameter in print():** By default Python's print() function ends with a newline(\n) which can be modified and made to any character using the 'end' parameter.

```
print("Hello")                  Output:  Hello
print("python")                          python

print("Hello", end = " ")       Output:  Hello python. ^_^
print("python", end = ". ^_^ ")

print("Hello", "python", end = ".", sep = ", ")       Output:  Hello, python.
```

❖ **Formatting using f-string**

```
x = 5
print(f"value of x is {x}")       Output:  value of x is 5
print(f"value of x is {x:5d}")    Output:  value of x is     5
# Above print statement prints the value of the variable x with a field width of 5 characters, right-aligned. 'd' represents decimal number, 'f' represents float

pi = 22/7
print(f"value of pi is {pi}")       Output:  value of pi is 3.142857142857143
print(f"value of pi is {pi:.3f}")   Output:  value of pi is 3.143
# If we want a float number to nth decimal place we've to write '.nf'
print(f"value of pi is {pi:8.3f}")  Output:  value of pi is    3.143
```

❖ **Formatting using Modulo Operator (%)**

```
x, pi = 5, 22/7

print("x = %d, pi = %f" % (x, pi))       Output:  x = 5, pi = 3.142857
# %d is replaced by 5 and %f is replaced by 22/7. Here, floating point number is printed up to 6 decimal places
```

```
print("x = %5d, pi = %.4f" % (x, pi))          Output:   x =     5, pi = 3.1429
```
#.4 is used to limit the decimal places of pi to 4

#5d, This specifies that the integer will take up at least 5 characters. If x has fewer digits, it will be padded with spaces on the left.

```
print("pi = %8.3f" % pi)                       Output:   pi =    3.143
```
#8 after mod, This specifies that the integer will take up at least 8 characters. If it has fewer digits, it will be padded with spaces on the left.

❖ Formatting using Format Method

```
pi = 22/7
```

```
print("x = {}, pi = {}".format(5, pi))         Output:   x = 5, pi = 3.142857142857143
```
#the first {} is given the first argument .i.e 5 and the second {} took the value or variable pi

```
print("x = {0}, pi = {1}".format(5, pi))       Output:   x = 5, pi = 3.142857142857143
print("pi = {1}, x = {0}".format(5, pi))       Output:   pi = 3.142857142857143, x = 5
```
# {0}, {1} represent the first and the second argument passed to the method.

```
print("x = {0:5d}, pi = {1:[illegible].4f}".format(5, pi))      Output:   x =     5, pi = 3.1429

print("pi = {0:8.3f}".format(pi))                     Output:   pi =    3.143
```
# In {0:8.3f}, **'0'** before ':' represents the argument no., as pi being 0th argument.

# **'8'** after ':' specifies that integer will take up at least 8 characters. If it has fewer digits, it will be padded with spaces on the left.

# **'.3f'** for the 3 decimal places

```
print("x = {x}, pi = {0:.4f}".format(pi, x = 5))      Output:   x = 5, pi = 3.1429
```

WEEK 4

Lists: An ordered and mutable collection of items. (`my_list = ['apple', 5, True]`)

- Empty list: `list()` or [ ] (square brackets)
- `type(my_list)` -> `<class 'list'>`
- #0 based indexing. (`my_list[0] = 'apple'` ; `my_list[1] = 5` ; `my_list[2] = True`)

List Methods: **click on method names for detailed description of each method.*

Method	Description
`list.append(x)`	Adds an element to the end of the list. Equivalent to: a[len(a):] = [x]
`list.insert(i, x)`	Insert item \`x\` at index \`i\`.
`list.extend(iterable)`	Extend the list by appending all the elements of the iterable. Equivalent to: a[len(a):] = iterable
`list.remove(x)`	Removes first occurrence of element \`x\` from the list. Raises ValueError if \`x\` not in list.
`list.pop(i)`	Removes the item from index \`i\` and returns it. If no index specified then -1 is taken as default value for \`i\`.
`list.clear()`	Removes all elements from list. Equivalent to: del a[:]
`list.copy()`	Return a shallow copy of the list. Equivalent to: a[:]
`list.index(x)`	Return the index of first occurrence of \`x\`. Raises ValueError if \`x\` not in list.
`list.count(x)`	Return number of times \`x\` appeared in list. (0 if \`x\` not in list)
`list.sort(reverse = True/False)`	Sort items of list in place, both numerically and alphabetically. ● Descending order if \`reverse = True\` ● Ascending order if \`reverse = False\` (default parameter)
`list.reverse()`	Reverse the items of list in place.

Popular List Functions:

Function	Description	Examples
len(list)	Returns the number of items in list.	a = [0, 1, 2, 3] #len(a) is 4
max(list)	Returns the largest item in the list.	a = [13, 32, -2, 5] #max(a) is 32
min(list)	Returns the smallest item in the list.	a = [13, 32, -2, 5] #min(a) is -2
sum(list)	Returns sum of all the elements of a list of numbers.	a = [13, 32, -2, 5] #sum(a) is 48
sorted(list, reverse = True/False)	Returns a new list with all elements of original list in: ● ascending order: (default) or if reverse = False ● Descending order: if reverse = True	#a = [13, 32, -2, 5] #sorted(a) = [-2, 5, 13, 32] or #sorted(a, reverse=False) = [-2, 5, 13, 32] #sorted(a, reverse=True) = [32, 13, 5, -2]
reversed(list)	Returns a list_reverse_iterator. *list(reversed(list)) can be used to get a new list containing all elements of original list in original order.	#a = [13, 32, -2, 5] #list(reversed(a)) = [5, -2, 32, 13] *note: do not forget to typecast reversed(a) to list datatype, as `reversed` doesn't return list data type.

- **Slicing a list:** list[start : end : step]   # a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# a[0:8:2] -> [0, 2, 4, 6]

- **List Concatenation:** # a = [1, 2, 3] b= [4, 5, 6]
  # a + b -> [1, 2, 3, 4, 5, 6] b + a -> [4, 5, 6, 1, 2, 3]

- **List Replication:** # a = [1, 2]
  # a*3 -> [1, 2, 1, 2, 1, 2]

- **Deleting a slice:**`del` deletes an element or slice (by using index)
  # a = [1, 2, 3, 4, 5, 6] # b = [1, 2, 3, 4, 5, 6]
  # del a[1] (delete elem from index1) # del b[1:5:2] (will delete [2, 4])
  # a -> [1, ,3 4, 5, 6] # b -> [1, 3, 5, 6]

- **Equality of lists:** Two lists are said to be equal if their corresponding elements are equal.
  ```
  [1, 2, 3] == [1, 2, 3] -> True
  [1, 2] ==  [1, 2, 3, 4] -> False
  ```

- **List Comparison:** Element wise comparison until a difference is found.
  ```
  [4, 3, 5, 2]  > [4, 2, 8, 9]  -> True
  ```
  (as 3 > 2, we'll not check further)

- **in operator** -> (Syntax: item in list): Returns True if item is present in list else False.

- A Matrix is represented as a list of lists. (# M = [[1, 2, 3], [4, 5, 6]])
  (no_of_rows = len(M), no_of_col = len(M[0])) (# M is a matrix of order: 2 X 3)

- Lists are Mutable (i.e. can be updated in place.)

- **Join Method:** Concatenates elements of a ***list of strings*** to a **string** with a user-defined `seperator_string` in between.

  Syntax: `seperator_string.join(iterable)`

  ```
  Eg. L = [‘I’, ‘like’, ‘apples’]
      word = ‘ ’.join(L)          #word -> ‘I like apples’
  ```

```
L = [1, 2, ‘and’, 4, 5]
L[2] = 3
print(L)

Output -> [1, 2, 3, 4, 5]
```

Tuples: An ordered and immutable collection of items. (`my_tup = ('apple', 5, True)`)

- Empty Tuple: `tuple()` or ( ) (parentheses)
- `type(my_tup)` -> `<class 'tuple'>`
- #0 based indexing. (`my_tup[0] = 'apple' ; my_tup[1] = 5 ; my_tup[2] = True`)
- Singleton Tuple: `(1, )` #{ not `(1)` }

- **Tuple Methods:**

Method	Description	Example
tuple.count(x)	Returns no. of times `x` appeared in the tuple.	# t = (1, 2, 1, 4, 5, 6, 11) # t.count(1) -> 2
tuple.index(x)	Returns the index of first occurrence of `x` in tuple.	# t = (1, 2, 1, 4, 5, 6, 11) # t.index(1) -> 0

- Few Functions Used With Tuples: • `sum, max, min, len`

- Tuple Comparison is similar to List Comparison only.

- **Slicing a tuple:** tuple[start : end : step] # a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
  # a[0:8:2] -> (0, 2, 4, 6)

- **Tuple Concatenation:** # a = (1, 2, 3) b= (4, 5, 6)
  # a + b -> (1, 2, 3, 4, 5, 6) b + a -> (4, 5, 6, 1, 2, 3)

- **Tuple Replication:** # a = (1, 2)
  # a*3 -> (1, 2, 1, 2, 1, 2)

- **`` `in` operator:``** Returns True if item is present in list else False.

Set: An unordered and mutable collection of items. (`my_set = {'apple', 5, True}`)

- Empty Set: `set()`
- `type(my_set)` -> `<class 'set'>`

Valid Elements of Sets:

- Any immutable and hashable objects can be used as elements of sets. Eg. int, tuple, string.
- Mutable objects like lists, sets and dictionaries cannot be used as elements of list.
- Note -> item = (1, 2, [3, 4], 5)   Here a is a tuple but still non-hashable as it contains a list which is mutable. Hence `item` can't be an element of a set.

Set Operations:

- Let's take set1 = {1, 2, 3, 4} and set2 = {1, 3, 5}

Operation	Expression	Example
Union	`set1 | set2`	{1, 2, 3, 4, 5}
Intersection	`set1 & set2`	{1, 3}
Set Difference	`set1 - set2`	{2, 4}
Symmetric Difference	`set1 ^ set2`	{2, 4, 5}
Subset	`set1 <= set2` (Is set1 subset of set2?)	False #eg. {1, 2, 3} <= {1, 2, 3} : True
Proper Subset	`set1 < set2`	False #eg. {1, 2} < {1, 2, 3} : True
Superset	`set1 >= set2` (Is set1 a superset of set2?)	False #eg. {1, 2, 3} >= {1, 2, 3} : True
Proper Superset	`set1 > set2`	False #eg. {1, 2} > {1} : True

Set Methods:

*click on the methods to see the detailed description.

Methods	Description
`set.add(x)`	Adds element `x` to the set.

`set.update(iterable)`	Adds all the elements of the `iterable` to set.
`set.remove(x)`	Removes element `x` from set. Raises error if element not found.
`set.discard(x)`	Removes element `x` from set. Do not raise any error if element not found.
`set.pop()`	Removes and returns a random element from set. Error in case of empty set.
`set.clear()`	Removes all elements from set.
`set.copy()`	Returns a shallow copy of set.
`set1.union(set2)`	Returns a new set which is union of set1 and set2.
`set1.intersection(set2)`	Returns a new set which is intersection of set1 and set2.
`set1.difference(set2)`	Returns a new set which is equivalent to set1 - set2
`set1.difference_update(set2)`	Removes all the elements of set1 which are present in set2.
`set1.symmetric_difference(set2)`	Returns a new set which is equivalent to set1 ^ set2
`set1.symmetric_difference_update(set2)`	Update the set1 keeping the elements found in either of set1 and set2, but not in both set1 and set2
`set1.issubset(set2)`	Returns `True` if set1 is subset of set2
`set1.issuperset(set2)`	Returns `True` is set1 is superset of set2
`set1.isdisjoint(set2)`	Returns `True` if both the set has no common elements.

- Few Functions Used With Sets: `len`, `sum`, `max`, `min`, `sorted`
- `in` operator checks the presence of an element quickly inside a set, as set uses hashing to find values faster compared to other collections.

Quick Summary of Collections

Data Type	Iterable	Collection	Indexable/ Sliciable	Ordered	Mutable	Uses Hashing
`str`	✓	✗	✓	✓	✗	✗
`range`	✓	✗	✗	✓	✗	✗
`list`	✓	✓	✓	✓	✓	✗
`tuple`	✓	✓	✓	✓	✗	✗
`set`	✓	✓	✗	✗	✓	✓

✓

Comprehensions

- **Using a single line `if` statement:**

  Syntax:

```
if condition:              ->        if condition: block
    block
```

```
Eg.
a = int(input())           ->       a = int(input())
if a%2 == 0:                        if a%2 == 0: print('even')
    print('even')
```

✓

- **Using a single line `if - else` statement.**

  Syntax:

```
if condition:
    block1            ->        block1 if condition else block2
else:
    block2
```

```
Eg.
a = int(input())                    a = int(input())
if a%2 == 0:          ->            print('even') if a%2 == 0 else print('odd')
     print('even')
else:
     print('odd')
```

- **Using single line ~~`while`~~ for loop:**

Syntax:

```
for x in iterable:   ->   for x in iterable: line1; line2
     line1
     line2
```

```
Eg.
for i in range(5):   ->   for i in range(5): print(i)
     print(i)
```

- **Using single line `while` loop:**

Syntax:

```
while condition:   ->     while condition: line1; line2
     line1
     line2
```

```
Eg.

i = 0                         i = 0

while i<5:               ->   while i<5: print(i); i += 1
     print(i)
     i += 1
```

List Comprehensions

- **Mapping a list:**

  Syntax:

```
L = [ ]                   ->  L = [f(x) for x in iterable]
for x in iterable:
    L.append(f(x))
```

  Eg. Create a new list whose elements are square of that of L1.

```
L1 = [1, 2, 3, 4]         ->   L1 = [1, 2, 3, 4]
L = [ ]                        L = [x**2 for x in L1]
for x in L1:
    L.append(x**2)
```

- **Filtering and Mapping (only using if with comprehension):**

  Syntax:

```
L = [ ]             ->   L = [f(x) for x in iterable if condition]
for x in iterable:
    if condition:
        L.append(f(x))
```

  Eg. Create a new list with only even elements of L1.

```
L = [ ]                 ->   L = [x for x in L1 if x%2 == 0]
for x in L1:
     if x%2 == 0:
          L.append(x)
```

- **if - else in List Comprehensions:**

  Syntax:

```
L = [ ]
for x in iterable:
     if condition:
          L.append(f(x))
     else:
          L.append(g(x))

->  L = [f(x) if condition else g(x) for x in iterable]
```

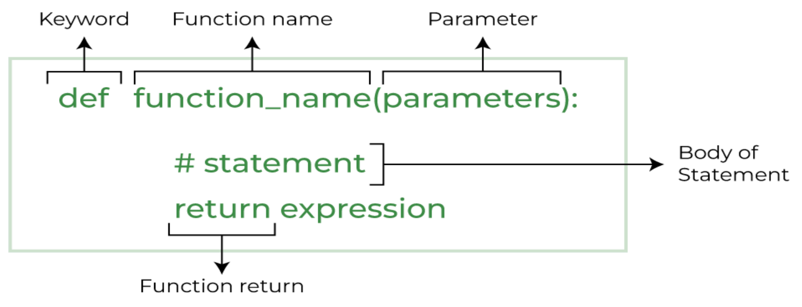
```
Eg. Create a new list by doubling the elements at even indices and squaring the elements at odd indices.

L = [ ]
for i in range(len(L1)):
     if i%2 == 0:
          L.append(L1[i]**2)
     else:
          L.append(L1[i]*2)


->   L = [L[i]**2 if i%2 == 0 else L[i]*2 for i in range(len(L1))]
```

Functions

■ Function in Python

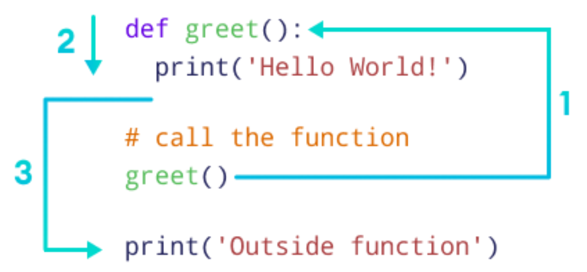
A Function is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code repeatedly for different inputs, we can do the function calls to reuse the code in it repeatedly.

Syntax:



```
def  function_name(parameters):
        # statement
        return expression
```

Calling a function:



```
def greet():
  print('Hello World!')

# call the function
greet()

print('Outside function')
```

1. When the function greet() is called, the program's control transfers to the function definition.
2. All the code inside the function is executed.
3. The control of the program jumps to the next statement after the function call.

❖ There are three types of functions:

- ➢ **Built-in function:** These are Standard functions in Python that are available to use. Example: `abs()`, `len()`, `max()`, `min()`, `etc.`
- ➢ **Library functions:** The functions that can be accessed by importing a library. Eg. `math.sqrt()`, `random.choice()`, `etc.`
- ➢ **User-defined function:** We can create our own functions based on our requirements.

❖ We return a value from the function using the return statement.

❖ **Arguments:** Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

Example:

```
def greet(name):
    print("Hello,", name)

greet("Python")    # Output: Hello, Python
```

❖ **Default Arguments:** A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def add(x, y=100):
    return x + y

print(add(10, 10))          # Output: 20
print(add(10))              # Output: 110
```

The add function is defined with two parameters: x and y, where y has a default value of 100. When add(10, 10) is called, it explicitly passes values for both x and y, resulting in x + y which computes to 20. When add(10) is called without specifying y, the default value of y=100 is used, resulting in x + y which computes to 110.

❖ **Keyword Arguments:** The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

❖ **Positional Arguments:** We can pass the value without mentioning the parameter name but at that time the positions of the passed values matter. During the function call the first value is assigned to the first parameter and the second value is assigned to the second parameter and so on. By changing the position, or if you forget the order of the positions, function gives unexpected result.

Example:

```
def sub(x, y):
    return x - y

print(sub(x=30, y=10))        # Output: 20
print(sub(y=10, x=30))        # Output: 20
print(sub(30, 10))            # Output: 20
print(sub(10, 30))            # Output: 20
```

The **sub()** function is defined to subtract **y** from **x**. Python allows function arguments to be passed by position or by keyword. In the given examples:

add(x=30, y=10) explicitly specifies the values for x and y, resulting in 30 - 10, which evaluates to 20.

add(y=10, x=30) uses keyword arguments, where the order of x and y is reversed compared to the function definition but still results in 30 - 10, also evaluating to 20 because of keyword arguments.

add(30, 10) and add(10, 30) both pass arguments by position. In the first case, the function computes 30 - 10, resulting in 20, but in the second case, the function returns 10 - 30, i.e., -20. As the argument name is not mentioned, it considers the first value as x and the second value as y.

❖ **Returning multiple values:** Python functions can return multiple values using one return statement. All values that should be returned are listed after the return keyword and are separated by commas.

```
def square_point(x, y, z):
    x_squared = x * x
    y_squared = y * y
    z_squared = z * z
    return x_squared, y_squared, z_squared                # Return all three values

three_squared, four_squared, five_squared = square_point(3, 4, 5)
print(three_squared, four_squared, five_squared)       # output: 9 16 2
```

Dictionary in Python

Basics of Dictionary

- Dictionaries in Python is a data structure that stores values in key: value format.
- A dictionary can be created by placing a sequence of elements within curly {} braces, separated by a 'comma'.
- A dictionary can also be created by the built-in function dict().

Example:

```python
    # creating an empty dictionary
    Dict = {}
    Dict = dict()

    # creating a dictionary in various ways
    Dict = {  1: 'Python',   2: 'dictionary',   3: 'example' }
    Dict = dict({1: 'Python',   2: 'dictionary',   3: 'example'})
    Dict = dict([(1, 'Python'), (2, 'dictionary'), (3, 'example')])
```

- Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated.
- Dictionary keys are case-sensitive, the same name but different cases of Key will be treated distinctly.

Example:

```python
Dict = { 1: 'Hello', 2: 123, 3: [32, 43, 12], 4: 123 }
# here values are different data types and also can be repeated

Dict = { 'py': 123, 'Py': 1234 }
# here, 'py' and 'Py' both are keys as keys are case-sensitive

Dict = {  1: 'Python',   1: 'dictionary',   3: 'example' }
print(Dict)          # {1: 'dictionary', 3: 'example'}
# while initialising a dictionary with same key, it always stores the latest value
```

■ Adding & Updating Elements to a Dictionary

- One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'.
- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.
- Updating an existing value in a Dictionary can be done by using the built-in update() method. Nested key values can also be added to an existing Dictionary.

Example:

Python
```
Dict = {}   # Empty Dictionary

Dict[0] = 'Hello'
Dict[1] = 2
Dict['Value_set'] = 2, 3, 4
print(Dict)

Dict[1] = 'Changed'     # Updated key value
Dict.update({0:5})     # Update using update method
print(Dict)
```

Output:

```
{0: 'Hello', 1: 2, 'Value_set': (2, 3, 4)}
{0: 5, 1: 'Changed', 'Value_set': (2, 3, 4)}
```

■ Accessing and Deleting Elements to a Dictionary

- We can access the dictionary items by using keys inside square brackets.
- There is also a method called get() to access the dictionary items. This method accepts the key as an argument and returns the value.
- The items of the dictionary can be deleted by using the del keyword.

Example:

Python
```
Dict = {1: 'Hello', 'name': 'Python', 3: 'World'}

# Accessing an element using key
print(Dict['name'])     # output: Python
# Accessing an element using get method
print(Dict.get(3))# output: World

del(Dict[1])# delete an element
print(Dict) # output: {'name': 'Python', 3: 'World'}
```

Nested Dictionary

A dictionary can be stored as the value of another dictionary.

Example:

Python
```python
Dict = {1: 'IIT', 2: 'Madras',
        3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}
}
print(Dict)
# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}

# accessing element of nested dictionary
print(Dict[3]['A'])     # output: Welcome

# updating & adding element of nested dictionary
Dict[3]['B'] = "Into"   # updating
Dict[3]['D'] = "World"  # adding
print(Dict)
# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'Into', 'C': 'Python', 'D': 'World'}}

# accessing element of nested dictionary
del(Dict[3]['D'])
print(Dict)
# {1: 'IIT', 2: 'Madras', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Python'}}
```

Dictionary Methods

Method	Description
clear()	Removes all items from the dictionary
copy()	Returns a shallow copy of the dictionary
get(key, default = value)	Returns the value of the specified key. If the key is not present in the dictionary it returns the default value if any default value is passed
items()	Returns a list containing a tuple for each key-value pair
keys()	Returns a list containing the dictionary's keys
values()	Returns a list of all the values of dictionary

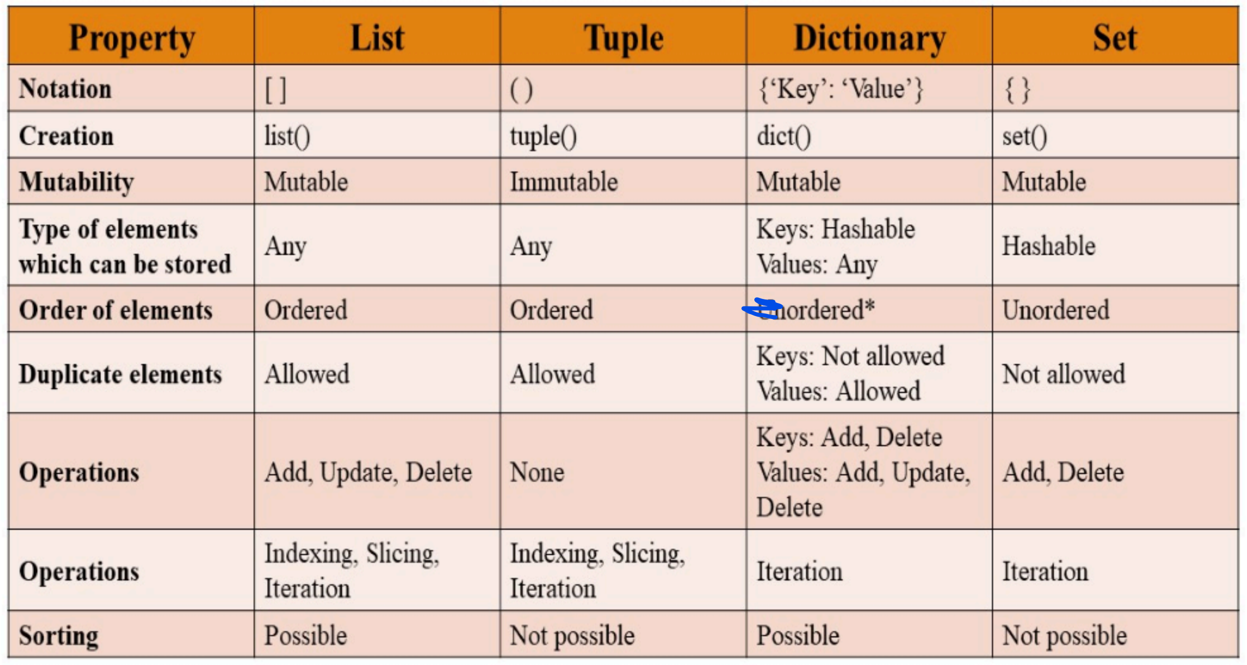
pop()	Remove the element with specified key
popitem()	Removes the last inserted key-value pair from the dictionary and returns it as a tuple
fromkeys()	Creates a new dictionary from the given sequence with the specific value
setdefault()	Returns the value of a key if the key is in the dictionary else inserts the key with a value to the dictionary
update()	Updates the dictionary with the elements from another dictionary or an iterable of key-value pairs. With this method, you can include new data or merge it with existing dictionary entries

Click on the method names to get more information.

■ Examples of important Dictionary Methods

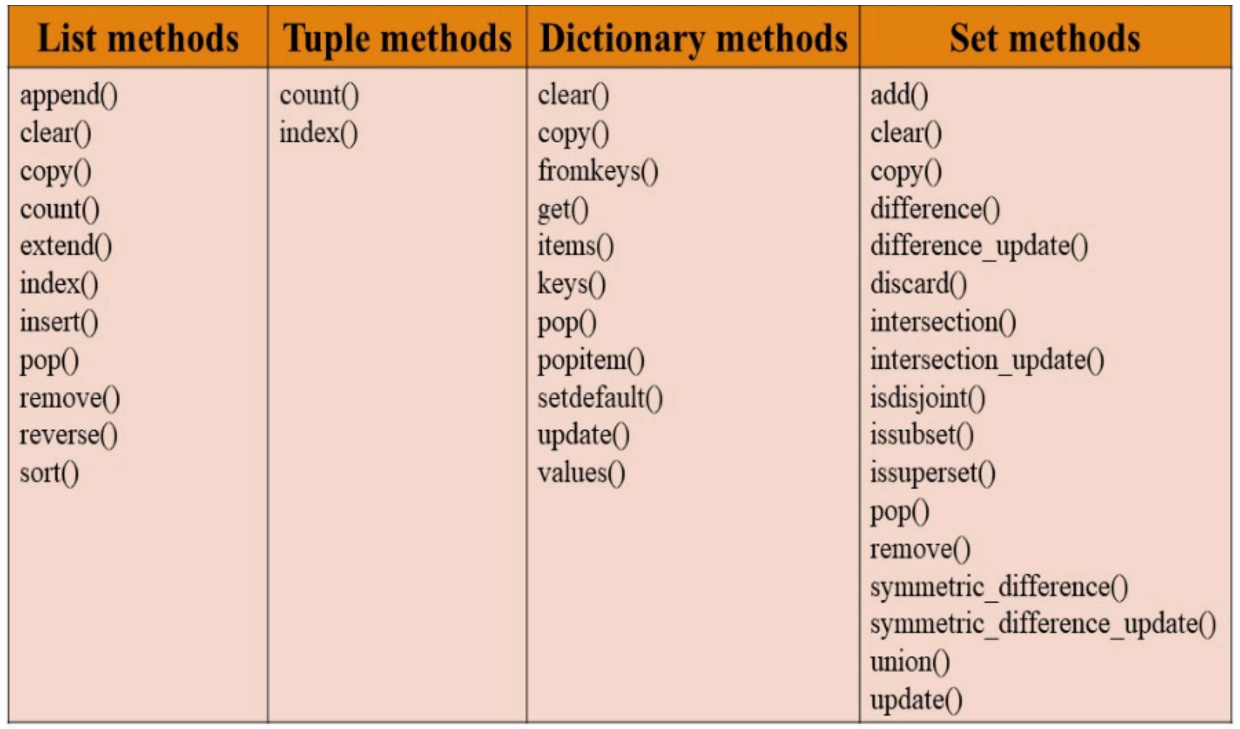
```python
d = {1: '001', 2: '010', 3: '011'}

print(d.get(2, "Not found"))    # 010, as 2 is a key
print(d.get(4, "Not found"))   # Not found, as 4 not present

print(d.items())
# dict_items([(1, '001'), (2, '010'), (3, '011')])

print(d.keys())
# dict_keys([1, 2, 3])

print(d.values())
# dict_values(['001', '010', '011'])

print(d.pop(1))     # 001
print(d)            # {2: '010', 3: '011'}
```

■ Summary of Python Collections



Property	List	Tuple	Dictionary	Set
Notation	[ ]	( )	{'Key': 'Value'}	{ }
Creation	list()	tuple()	dict()	set()
Mutability	Mutable	Immutable	Mutable	Mutable
Type of elements which can be stored	Any	Any	Keys: Hashable Values: Any	Hashable
Order of elements	Ordered	Ordered	Unordered*	Unordered
Duplicate elements	Allowed	Allowed	Keys: Not allowed Values: Allowed	Not allowed
Operations	Add, Update, Delete	None	Keys: Add, Delete Values: Add, Update, Delete	Add, Delete
Operations	Indexing, Slicing, Iteration	Indexing, Slicing, Iteration	Iteration	Iteration
Sorting	Possible	Not possible	Possible	Not possible

*Python 3.6 and earlier. Dictionaries are ordered as per Python 3.7 and above.



List methods	Tuple methods	Dictionary methods	Set methods
append()	count()	clear()	add()
clear()	index()	copy()	clear()
copy()		fromkeys()	copy()
count()		get()	difference()
extend()		items()	difference_update()
index()		keys()	discard()
insert()		pop()	intersection()
pop()		popitem()	intersection_update()
remove()		setdefault()	isdisjoint()
reverse()		update()	issubset()
sort()		values()	issuperset()
			pop()
			remove()
			symmetric_difference()
			symmetric_difference_update()
			union()
			update()

■ Dot Product in Python

Let two vectors (lists) are,

x = [x1, x2, x3, x4, ….. , xn]
y = [y1, y2, y3, y4, ….. , yn]

Dot product is defined as, $$s = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

Python
```python
x = [1, 2, 3, 4]
y = [5, 6, 7, 8]
dot_product = 0

for i in range(len(x)):
    dot_product += x[i]*y[i]

print(dot_product)      # 70
```

■ Multiply Two Matrices

Let, A be a matrix of dimension R1 x C1. Here R1 represents the row of A and C1 represents the column of A. It can be denoted as A [R1 X C1].

Let A [R1 X C1] and B [R2 X C2] are matrices. The multiplication of matrices can be done if and only if C1 = R2 and the resultant matrix becomes Z [R1 X C2].

Here is the code of Matrix multiplication:

Python
```python
def matrix_multiplication(X, Y):
    # row and columns of matrix X
    r1, c1 = len(X), len(X[0])

    # row and columns of matrix Y
    r2, c2 = len(Y), len(Y[0])

    # initializing result matrix
    # the dimension of resultant matrix: r1 x c2
    result = []
    for i in range(r1):
        result.append([0]*c2)
    # matrix multiplication
    # iterate through rows of X
    for i in range(r1):
       # iterate through columns of Y
       for j in range(c2):
           # iterate through rows of Y
           for k in range(r2):
               result[i][j] += X[i][k] * Y[k][j]

    return result


# 3x3 matrix
X = [
    [12,7,3],
    [4 ,5,6],
    [7 ,8,9]
]
# 3x4 matrix
Y = [
    [5,8,1,2],
    [6,7,3,0],
    [4,5,9,1]
]


result = matrix_multiplication(X,Y)
for r in result:
   print(r)


# output
# [114, 160, 60, 27]
# [74, 97, 73, 14]
# [119, 157, 112, 23]
```

■ Python Scope of Variables

Python Local variable: Local variables are initialised within a function and are unique to that function. It cannot be accessed outside of the function.

```
Python
def f():
    s = "I love Python"    # local variable
    print(s)

f()
print(s)     # this line throws an error as
             # local variable cannot be accessed outside of the function


# Output
# I love Python
# NameError: name 's' is not defined
```

Python Global variables: Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

```
Python
def f():
    print(s)

# global scope
s = "I love Python"
f()

# Output
# I love Python
```

Global and Local Variables with the Same Name

If a global variable is reinitialized inside a function. It is considered as a local variable of that function and if any modification is done on that variable, the value of the global variable will remain unaffected. For example:

```
Python
def f():
    s = "Hello World"
    print(s)
```

```
# global scope
s = "I love Python"
f()
print(s)

# Output
# Hello World
# I love Python
```

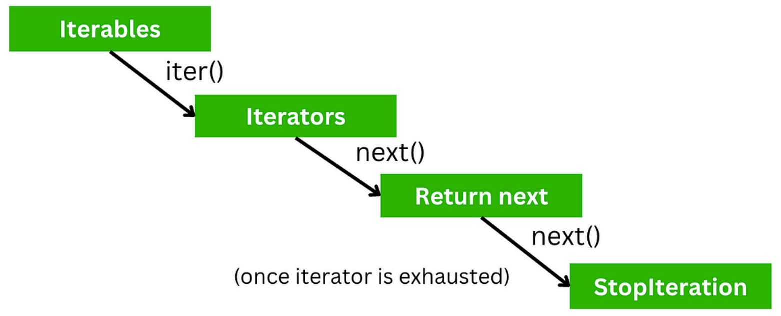
To modify the value of the global variable we need to use the **global** keyword.

```
Python
def f():
    global s
    s = "Hello World"
    print(s)

# global scope
s = "I love Python"
f()
print(s)

# Output
# Hello World
# Hello World
```

■ Iterators and Generators in Python

Iterator

In Python, an iterator is an object used to iterate over iterable objects such as lists, tuples, dictionaries, and sets. An object is called iterable if we can get an iterator from it or loop over it.



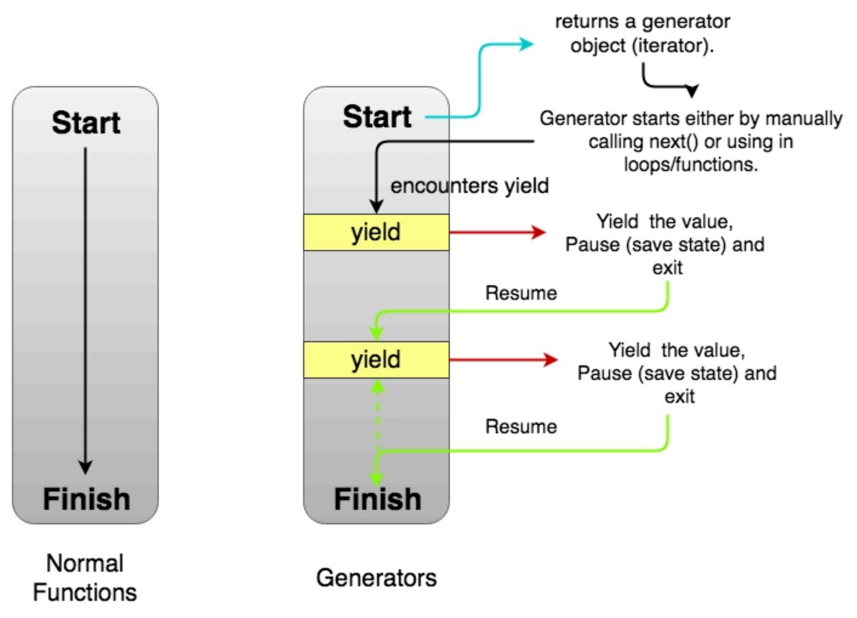
- **iter()** function is used to create an iterator containing an iterable object.
- **next()** function is used to call the next element in the iterable object.

```python
iter_list = iter(['I', 'Love', 'Python'])
print(next(iter_list))     # I
print(next(iter_list))     # Love
print(next(iter_list))     # Python
```

Generators

Python has a generator that allows you to create your iterator function. A generator is somewhat of a function that returns an iterator object with a succession of values rather than a single item. A yield statement, rather than a return statement, is used in a generator function.

The difference is that, although a return statement terminates a function completely, a yield statement pauses the function while storing all of its states and then continues from there on subsequent calls.


returns a generator object (iterator).
Start
Generator starts either by manually calling next() or using in loops/functions.
encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Start
Finish
Normal Functions
Generators

```python
def power(limit):
    x = 0
    while x<limit:
        yield x*x
        yield x*x*x
        x += 1

a = power(5)
print(next(a), next(a))    # 0 0
print(next(a), next(a))    # 1 1
print(next(a))            # 4
print(next(a))            # 8
print(next(a), next(a))   # 9 27
```

■ Python Lambda Function

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know.

Syntax: **lambda arguments: expression**

Example:

```
Python
cube = lambda y: y*y*y
print("cube:", cube(5))   # cube: 125
```

✓

■ enumerate() Function

The **enumerate()** function adds a counter to an iterable and returns it as an enumerate object (iterator with index and the value).

Syntax: **enumerate(iterable, start=0)**

- **iterable** - a sequence, an iterator, or objects that support iteration.
- **start** (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

Example:

```
Python
l1 = ["eat", "sleep", "repeat"]
s1 = "code"

obj1 = enumerate(l1)      # creating enumerate objects
obj2 = enumerate(s1)

print ("Return type:", type(obj1))     # Return type: <class 'enumerate'>
print (list(enumerate(l1)))      # [(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

# changing start index to 2 from 0
print (list(enumerate(s1, 2)))   # [(2, 'c'), (3, 'o'), (4, 'd'), (5, 'e')]
```

✓

■ zip() Function

The **zip()** function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.

Syntax: zip(iterable1, iterable2, iterable3, …..)

Example:

```
Python
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]

result = zip(languages, versions)
print(list(result))
# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

■ map() Function

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Syntax: map(function, iterables)

- **function** - a function that is applied to each element of an iterable.
- **iterables** - iterables such as lists, tuples, etc.

Example:

```
Python
def square(n):
    return n*n

numbers = (1, 2, 3, 4)
result = map(square, numbers)
print(result)       # Output: <map object at 0x7f722da129e8>
print(set(result))  # Output: {16, 1, 4, 9}

# using lambda function in map
print(list(map(lambda x: x*x, numbers)))   # Output: [1, 4, 9, 16]

num1 = [1, 2, 3]
num2 = [10, 20, 40]

# add corresponding items from the num1 and num2 lists
result = map(lambda n1, n2: n1+n2, num1, num2)
print(tuple(result))              # Output: (11, 22, 43)
```

■ filter() Function

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax: filter(function, iterables)

- **function** - a function that is applied to each element of an iterable.
- **iterables** - iterables such as lists, tuples, etc.

Example:

```
Python
def check_even(number):
    return number % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# if an element passed to check_even() returns True, select it
even_numbers_iterator = filter(check_even, numbers)
print(even_numbers_iterator)      # Output: <filter object at 0x7715fbaec0d0>

# converting to list
print(list(even_numbers_iterator))    # Output: [2, 4, 6, 8, 10]



# using lambda function
print(list(filter(lambda num: num%2==0, numbers)))
# Output: [2, 4, 6, 8, 10]
```

WEEK 8

File Handling in Python

■ Open a File

We can open a file in various modes in Python.

```
Python
# syntax to open a file
file = open(file_path, mode)
```

Important Modes are:

Mode	Description
r	open an existing file for a read operation. If the file does not exist, it will throw an error.
w	open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.
a	open an existing file for append operation. It won't override existing data. If the file is not present then it creates the file.
r+	To read and write data into the file. This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
w+	To write and read data. It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
a+	To append and read data from the file. It won't override existing data.

Consider the 'example.txt' file exists with the following contents:

```
Unset
Hello
How are you?
Hope you are enjoying Python.
```

■ Working in Read Mode

Example 1: Here we are reading a file and printing each line.

```python
file = open('example.txt', 'r')

for line in file:
    print(line)

file.close()        # it is a good practice to close a file


# Output
'''
Hello

How are you?

Hope you are enjoying Python.

'''
```

Here you can notice there is an extra line after each of the lines. Because in the file system, there is a **'\n'** character that gets included when we press enter to go to the new line to write.

Example 2: Here we are going to read a file and store all the lines in a list format.

```python
file = open('example.txt', 'r')

# readlines() method reads all the lines of the file and
# returns in a list format where every line is an item of the list
l = file.readlines()
print(l)

file.close()

# output
# ['Hello\n', 'How are you?\n', 'Hope you are enjoying Python.\n']
```

Here you can observe the '\n' character present at the end of the lines.
Example 3: Here we are going to read one line at a time.

```python
file = open('example.txt', 'r')

# readline() method reads a line from a file and returns it as a string.
first_line = file.readline()
```

```
print(first_line.strip())
# to get rid of that extra '\n' character strip() method is used

second_line = file.readline()
print(second_line.strip())

# we can print directly without storing it
print(file.readline())

file.close()

# output
'''
Hello
How are you?
Hope you are enjoying Python.
'''
```

Example 4: we will extract a string that contains all characters in the Python file then we can use the read() method.

```
Python
file = open('example.txt', 'r')

print(file.read())

file.close()

# output

'''
Hello
How are you?
Hope you are enjoying Python.

'''
```

Example 4: seek() and tell() methods.

seek() method is used to change the position of the File Handle to a given specific position. The file handle is like a cursor, which defines where the data has to be read or written in the file.

tell() method prints the current position of the File Handle.

Syntax of seek() method:

```
Unset
Syntax: f.seek(offset, from_what), where f is file pointer

Parameters:

Offset: Number of positions to move forward

from_what: It defines point of reference.

Returns: Return the new absolute position.
```

The reference point is selected by the from_what argument. It accepts three values:
0: sets the reference point at the beginning of the file
1: sets the reference point at the current file position
2: sets the reference point at the end of the file
By default from_what argument is set to 0.

```python
# for this case assume the 'example.txt' contains the following line
# Code is like humor. When you have to explain it, it's bad.

f = open("example.txt", "r")

# Second parameter is by default 0 sets Reference point to twentieth
# index position from the beginning
f.seek(20)

# prints current position
print(f.tell())

print(f.readline())
f.close()


# Output
'''
20
When you have to explain it, it's bad.
'''
```

■ Working in Write Mode

Example 1: Here we are writing a file.

```python
file = open('example.txt', 'w')

file.write('new line 1')
file.write('new line 2')

file.close()

# The content of the file becomes
'''
new line 1new line 2
'''
```

write() method overrides the existing file if exists otherwise creates a new file. You can observe two lines added into a single line, if you want to add it in a different line we have to mention the '\n' character whenever you want to break a line.

```python
file = open('example.txt', 'w')

file.write('new line 1\n')
file.write('new line 2\n')

file.close()

# The content of the file becomes
'''
new line 1
new line 2

'''
```

Example 2: Write multiple strings at a time.

```python
file = open('example.txt', 'w')

file.writelines(["Item 1\n", "Item 2 ", "Item 3\n", "Item 4"])

file.close()

# The content of the file becomes
'''
Item 1
Item 2 Item 3
Item 4
'''
```

Working in Append Mode

Example: We append a new line a the end of the file.

Python
```python
file = open('example.txt', 'a')

file.write('a new line will be added')

file.close()

# The content of the file becomes

'''
Hello
How are you?
Hope you are enjoying Python.
a new line will be added
'''
```

Caesar Cipher

Encrypt the message

Python
```python
import string

def encrypt(s, shift):
    lower_case = list(string.ascii_lowercase)
    upper_case = list(string.ascii_uppercase)

    encrypt_msg = ''
    for char in s:
        if char in lower_case:
            encrypt_msg += lower_case[(lower_case.index(char) + shift) % 26]
        elif char in upper_case:
            encrypt_msg += upper_case[(upper_case.index(char) + shift) % 26]
        else:
            encrypt_msg += char

    return encrypt_msg


print(encrypt("Hello, King Caesar!!!", 3))
```

This Python code defines a function `encrypt` that performs a Caesar Cipher encryption on a given string `s` with a specified `shift` value. The function first creates lists of lowercase and uppercase alphabet letters. It then initializes an empty string `encrypt_msg` to store the encrypted message. As it iterates through each character in the input string, it checks if the character is a lowercase or uppercase letter. If so, it shifts the character by the specified `shift` value within the bounds of the alphabet and appends the shifted character to `encrypt_msg`. If the character is not a letter, it appends the character as is. Finally, it returns the encrypted message. The example given shifts each letter in "Hello, King Caesar!!!" by 3 positions, resulting in "Khoor, Lqqj Fdhvdu!!!".

■ Decrypt the message

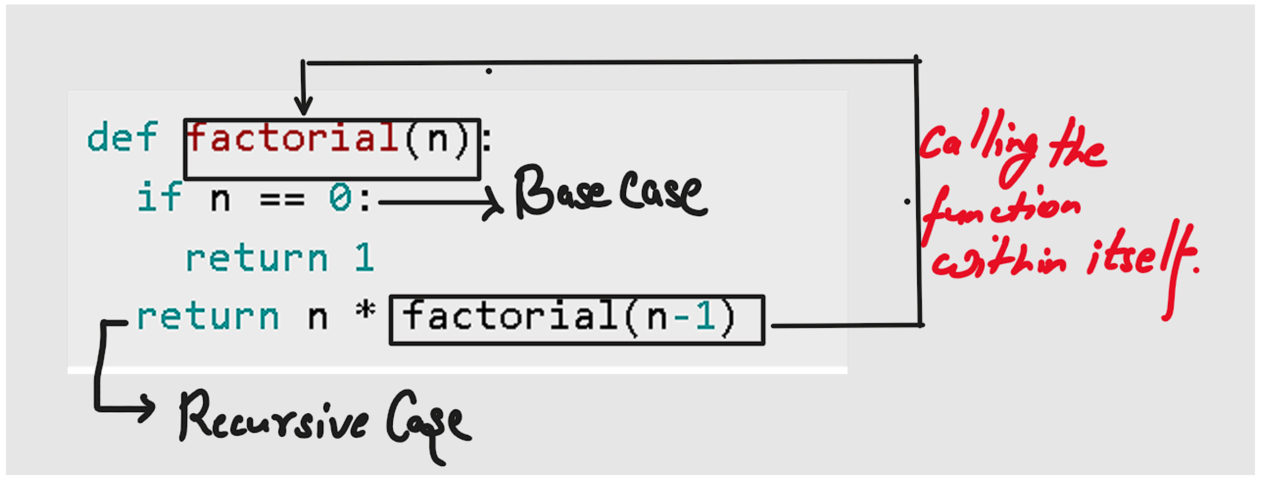
Python
```
import string

def decrypt(s, shift):
    lower_case = list(string.ascii_lowercase)
    upper_case = list(string.ascii_uppercase)

    decrypt_msg = ''
    for char in s:
        if char in lower_case:
            decrypt_msg += lower_case[(lower_case.index(char) - shift) % 26]
        elif char in upper_case:
            decrypt_msg += upper_case[(upper_case.index(char) - shift) % 26]
        else:
            decrypt_msg += char

    return decrypt_msg


print(decrypt("Khoor, Nlqj Fdhvdu!!!!!!", 3))
```

This Python code defines a function `decrypt` that performs a Caesar Cipher decryption on a given string `s` with a specified `shift` value. The function creates lists of lowercase and uppercase alphabet letters and initializes an empty string `decrypt_msg` to store the decrypted message. As it iterates through each character in the input string, it checks if the character is a lowercase or uppercase letter. If so, it shifts the character backwards by the specified `shift` value within the bounds of the alphabet and appends the shifted character to `decrypt_msg`. If the character is not a letter, it appends the character as is. Finally, it returns the decrypted message. The example given shifts each letter in "Khoor, Nlqj Fdhvdu!!!!!!" backwards by 3 positions, resulting in "Hello, King Caesar!!!!!!".

WEEK 9

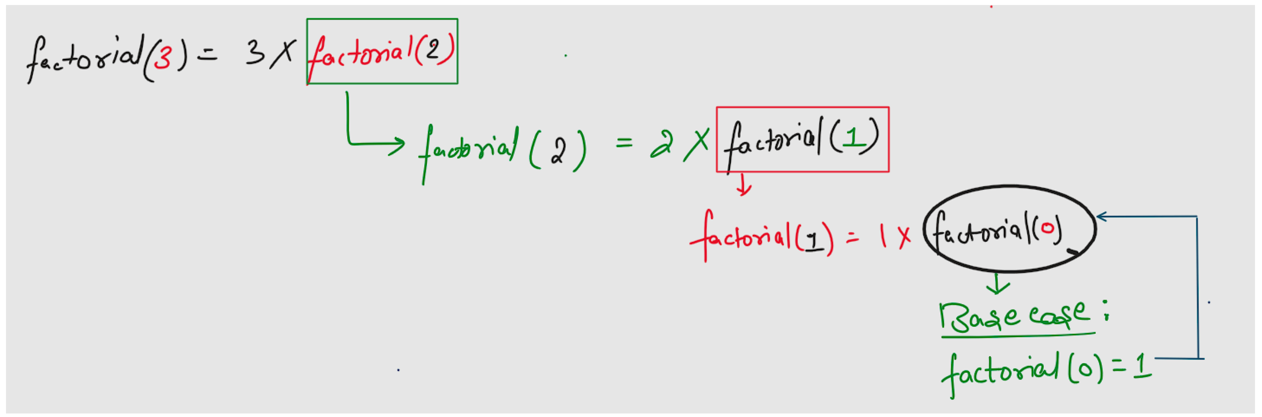
<u>Recursion</u>

Process of calling a function within itself.



```python
def factorial(n):
    if n == 0:
      return 1
    return n * factorial(n-1)
```

<u>Illustration of recursion:</u>

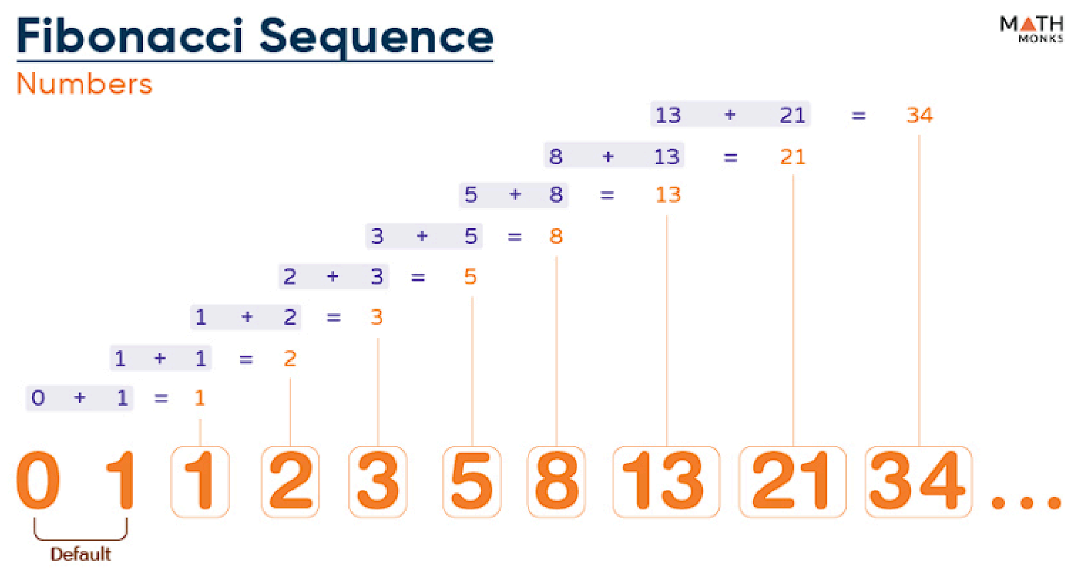


factorial(3) = 3 X factorial(2)
factorial(2) = 2 X factorial(1)
factorial(1) = 1 x factorial(0)
Base case:
factorial(0) = 1

■ **Fibonacci Sequence:**

Series where each number is obtained by adding its two preceding numbers, starting with 0 followed by 1.

$$F(n) = \begin{cases} 0 \ ; & n = 0 \\ 1 \ ; & n = 1 \\ F(n-1) + F(n-2) \ ; & n \geq 2 \end{cases}$$



<u>Recursive code for getting nth term of fibonacci series:</u>

```
def fibo(n):
  if n == 0:     # base case
    return 0
  if n == 1:     # base case
    return 1

  # recursive case
  return fibo(n-1) + fibo(n-2)
```

■ Sorting A List Using Recursion:

```
def recursive_sort(L):
  if L == [ ]:   # Base Case
    return L

  # Store minimum element of list in variable `mini`.
  mini = min(L)

  L.remove(mini) # Removes `minimum element` from the list.
```

```python
    # Recursive Case
    return [mini] + recursive_sort(L)
```

Binary Search

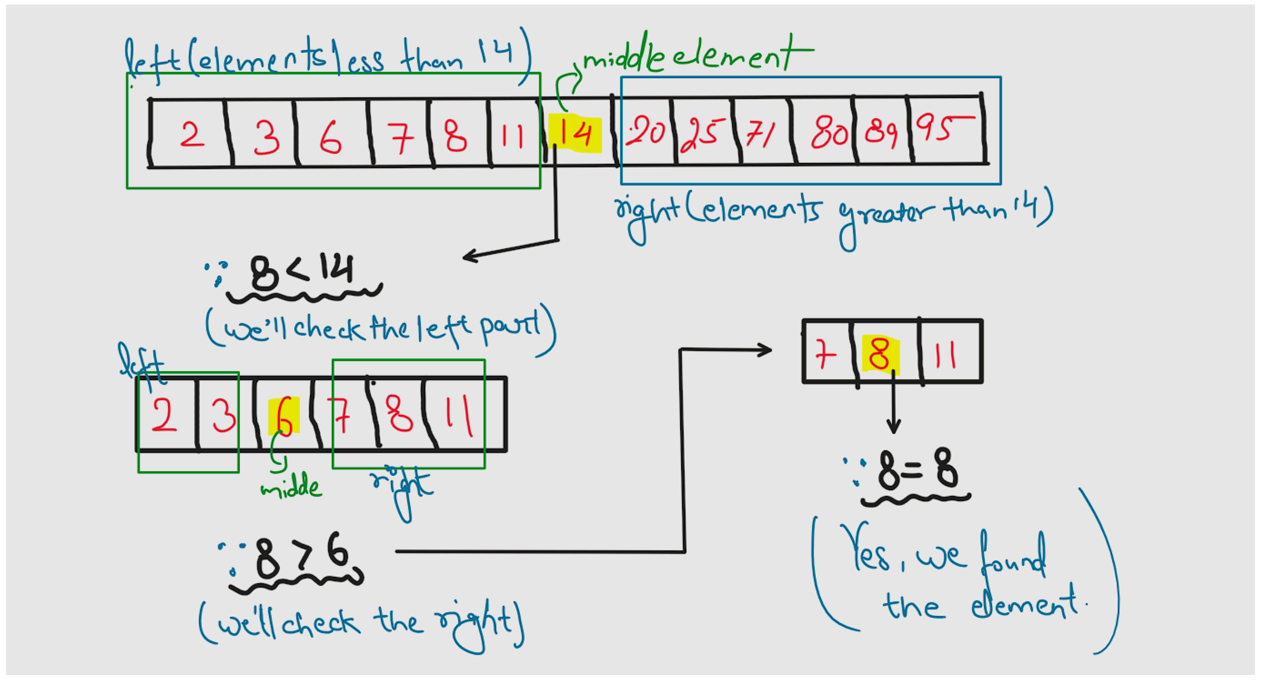
An efficient method to search the presence of an element in a sorted array.

Working:

1. Compare your target `x` with middle element.
2. If `x` == middle element we return `Element found`.
3. Else if it is **less than middle element**, then `x` can only lie in the ***left(smaller) half subarray***. So, we repeat the algorithm again in the left part.
4. Else if it is **greater than middle element**, we repeat the algorithm in the ***right part.***

   If the list becomes empty and we couldn't find the element, we return `Element not found.`

An Illustration to check presence of `8` in a sorted list:



- A real life example is when we search for a word in dictionary.
- **Note**: Binary Search is much more efficient than obvious search as we're halving the list every time.

■ An Iterative Code For Binary Search:

```
def binary_search(L, x):

  while L: # Would repeat the process till list becomes empty.
    mid = len(L)//2

    if L[mid] == x:
      return 'Element Found.'

    elif L[mid] > x:  # If `x` is less than middle element, then we'll check the left half of the list.
      L = L[:mid]

    elif L[mid] < x:  # If `x` is greater than middle element, then we'll check the right half of the list.
      L = L[mid+1:]

  # If we couldn't find the element and the list becomes empty.
  return 'Element Not Found.'
```

■ Binary Search Using Recursion:

```
def recursive_binary_search(L, x):
  mid = len(L)//2

  if not(L):                        # Base case for if element not in list (ie. list becomes empty).
    return 'Element Not Found'

  elif L[mid] > x:                 # Recursive Case (Checking the left part when `x` < middle element)
    return recursive_binary_search(L[:mid], x)

  elif L[mid] < x:                 # Recursive Case (Checking the right part when `x` > middle element)
    return recursive_binary_search(L[mid+1:], x)

  return 'Element Found'           # Base case for element found in list.
```

WEEK 10

Exception Handling in Python

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.

When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit. In Python, we catch exceptions and handle them using try and except code blocks.

❖ Syntax of try…..except block:

```
Python
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Example: Here we are trying to access the array element whose index is out of bound and handle the corresponding exception.

```
Python
a = [1, 2, 3]
try:
    print ("Fourth element = %d" %(a[3]))
except:
    print ("An error occurred")

# Output
'''
An error occurred
'''
```

As the exception is dealt here by the **try…exception** block, the output will be shown instead of abruptly stopping the program by showing an error message.

❖ Catching Specific Exceptions

For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently. Please be aware that only one handler will be run at a time.

- **Generic syntax:**

```
Python
try:
    # statement(s)
except Exception1:
    # statement(s)
except Exception2:
    # statement(s)
except:
    # ifnone of the above cases is matched
```

Example 1:

```
Python

try:
    print(5/0)

except IndexError:
    print("Index Out of Bound.")
except ZeroDivisionError:
    print("Denominator cannot be 0.")
except:
    print("Some other exception occurred")


# Output
'''
Denominator cannot be 0.
'''
```

The code attempts to divide 5 by 0, which raises a ZeroDivisionError. The try block is used to catch exceptions, and the except block handles specific errors. Since the error is a ZeroDivisionError, the corresponding except block is triggered, printing "Denominator cannot be 0." If any other exception occurs, it will be caught by the last generic except block, but in this case, it's not needed as the specific error has already been handled.

Example 2:

```
Python
try:
    print(l)

except IndexError:
    print("Index Out of Bound.")
except ZeroDivisionError:
    print("Denominator cannot be 0.")
```

```
# you can write the following line instead of just writing except
# to view the error message associated with the Exception
except Exception as e:
    print(e)


# Output
'''
name 'l' is not defined
'''
```

The code tries to print the value of the variable l, which is not defined, leading to a NameError. The try block is used to handle exceptions, and the generic except Exception as e block catches the NameError, printing the error message associated with it. The other specific except blocks for IndexError and ZeroDivisionError are not triggered since they don't match the raised exception.

❖ try block with else and finally block

- ❖ There are two optional blocks, i.e., the **else** block and **finally** block, associated with the **try** block.
- ❖ The **else** comes after the **except** block(s). Only when the try clause fails to throw an exception does the code go on to the **else** block. The **finally** block comes at the end.
- ❖ The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Example 1:

```
Python
try:
    print(5//0)

except ZeroDivisionError:
    print("Can't divide by zero")

else:
    print("This line is only gets executed when there is no exception")

finally:
    print('This line is always executed')


# Output
'''
Can't divide by zero
This is always executed
'''
```

The code attempts to perform integer division by zero (5//0), which raises a ZeroDivisionError. The except block catches this error and prints "Can't divide by zero." The else block, which would execute if no exception occurred, is skipped. The finally block is then executed regardless of whether an exception was raised, printing "This line is always executed."

Example 2:

```
Python
try:
    print(5//2)

except ZeroDivisionError:
    print("Can't divide by zero")

else:
    print("This line is only gets executed when there is no exception")

finally:
    print('This line is always executed')


# Output
'''
2
This line is only gets executed when there is no exception
This is always executed
'''
```

The code performs integer division ('5//2'), which successfully results in '2'. Since no exception is raised, the 'else' block executes, printing "This line is only gets executed when there is no exception." Finally, the 'finally' block executes, printing "This line is always executed," ensuring that it runs regardless of the outcome in the 'try' block.

❖ Some types of exceptions in Python:

In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError:** This exception is raised when a variable or function name is not found in the current scope.
- **IndexError:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError:** This exception is raised when a key is not found in a dictionary.
- **ValueError:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

- **ZeroDivisionError:** This exception is raised when an attempt is made to divide a number by zero.
- **ImportError:** This exception is raised when an import statement fails to find or load a module.
- **FileNotFoundError:** This exception is raised when the given file is not present in the current directory.

❖ Raising Exception

The raise statement allows the programmer to force a specific exception to occur. It specifies the exception that should be raised.

Example:

```python
x = 5
if x > 3:
    raise Exception("Value is greater than 3")

# Output
'''
Value is greater than 3
'''
```

The code checks if the variable x is greater than 3. Since x is 5, which satisfies the condition, an Exception is manually raised with the message "Value is greater than 3," terminating the program with this error message.

Class & Object in Python

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. The class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

An Object is an instance of a Class. We can create multiple objects of a class.

❖ Define Python Class and Object

```python
class ClassName:
    # class definition

# creating an object
obj1 = ClassName()
obj2 = ClassName()
```

❖ **Access attributes of Objects:** We use the '.' (dot) notation to access the attributes of a class.

❖ **Methods:** Objects can also contain methods. Methods in objects are functions that belong to the object.

❖ **__init__() method:** This method is known as the **constructor** or **object initializer** because it defines and sets the initial values for the object's attributes. It is executed at the time of Object creation. It runs as soon as an object of a class is instantiated.

Example:

```python
class Person:
  # constructor
  def __init__(self, name, age):
    self.name = name
    self.age = age

  # methods
  def display(self):
    print(self.name, self.age)


# creating object of the class
p1 = Person("John", 36)

# accessing the attribute of the object
print(p1.name)

# using the method of the object
p1.display()
```

```
# Output
'''
John
John 36
'''
```

The code defines a Person class with a constructor (__init__) that initializes name and age attributes when an object is created. The class also has a display method that prints the name and age of the person. An object p1 is created with the name "John" and age 36. The name attribute of p1 is accessed and printed, resulting in "John". Then, the display method is called, which prints "John 36".

- **self parameter:** The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
- **__str__() Method:** The __str__() function controls what should be returned when the class object is represented as a string. If the __str__() function is not set, the string representation of the object is returned.

```python
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"Name: {self.name}, age: {self.age}"


p1 = Person("John", 36)
print(p1)


# Output
'''
Name: John, age: 36
'''
```

The code defines a 'Person' class with a constructor ('__init__') that initializes the 'name' and 'age' attributes. It also defines a '__str__' method, which returns a formatted string when an instance of 'Person' is printed. When the object 'p1' is created with the name "John" and age 36, and 'print(p1)' is called, the '__str__' method is invoked, resulting in the output "Name: John, age: 36."

❖ Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self, whereas class variables are variables whose value is assigned in the class.

Example:

```python
class Student:
  # class variable
  count = 0

  def __init__(self, name, age):
    # instance variable
    self.name = name
    self.age = age
    Student.count += 1

  def __str__(self):
    return f"Name: {self.name}, age: {self.age}"


s1 = Student("John", 18)
print(s1)
print("Student count:", Student.count)

s2 = Student("Smith", 19)
s3 = Student("Alex", 17)
print(s2)
print(s3)
print("Student count:", Student.count)


# output
'''
Name: John, age: 18
Student count: 1
Name: Smith, age: 19
Name: Alex, age: 17
Student count: 3
'''
```

In the 'Student' class example, 'count' is a class variable shared by all instances of the class, used to track the total number of 'Student' objects created. Each time a new 'Student' instance is initialized, 'count' is incremented, reflecting the total number of students. In contrast, 'name' and 'age' are instance variables specific to each individual 'Student' object, storing unique attributes for each instance. While 'count' maintains a global state across all instances, 'name' and 'age' hold data specific to the object they belong to.

Inheritance in Python

- Inheritance allows you to inherit the properties of a class, i.e., parent class to another, i.e., child class.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

❖ General Syntax

Python
```python
Class ParentClass:
    {Body}

Class ChildClass(BaseClass):
    {Body}
```

Example:

Python
```python
class Person():
  def Display(self):
    print("From Person Class")

class Employee(Person):
  def Print(self):
    print("From Employee Class")

per = Person()
per.Display()

emp = Employee()
emp.Print()
emp.Display()

# output
'''
From Person Class
From Employee Class
From Person Class
'''
```

The code demonstrates inheritance in Python. The 'Person' class has a method 'Display' that prints "From Person Class." The 'Employee' class inherits from 'Person' and adds its own method 'Print' that prints "From Employee Class." An instance 'per' of 'Person' is created and calls the 'Display' method. Then, an instance 'emp' of 'Employee' is created, which calls both its own 'Print' method and the inherited 'Display' method from 'Person', demonstrating how 'Employee' can access methods from its parent class. The output reflects these method calls.

- **Method Overriding:** Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type (or sub-type) as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- **super() function:** The super() function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

Syntax:

Python
```python
# use parent class attribute
super().attributeName
# use parent class method
super().methodName1()      # without parameter
super().methodName2(parameters)  # with parameter


## Alternate way
# use parent class attribute
ParentClassName.attributeName
# use parent class method
ParentClassName.methodName1(self)       # without parameter
ParentClassName.methodName2(self, parameters)  # with parameter
```

Example 1:

Python
```python
class Person():
  def Display(self):
    print("From Person Class")

class Employee(Person):
  def Display(self):
    print("From Employee Class")

per = Person()
per.Display()

emp = Employee()
emp.Display()

# output
'''
From Person Class
From Employee Class
'''
```

In this example, the 'Employee' class inherits from the 'Person' class. Both classes have a 'Display' method, but the 'Employee' class overrides the 'Display' method from 'Person'. When 'per.Display()' is called on a 'Person' object, it executes the 'Display' method from 'Person', printing "From Person Class." When 'emp.Display()' is called on an 'Employee' object, it executes the overridden 'Display' method in 'Employee', printing "From Employee Class." This demonstrates that the subclass method takes precedence when called on an instance of the subclass.

Example 2:

```python
class Person():
  def Display(self):
    print("From Person Class")

class Employee(Person):
  def Display(self):
    print("From Employee Class")
    super().Display()
    # Person.Display(self)     # alternate way

per = Person()
per.Display()

emp = Employee()
emp.Display()

# output
'''
From Person Class
From Employee Class
From Person Class                ✓
'''
```

In the example, the 'Employee' class overrides the 'Display' method of the 'Person' class with its own version. When 'emp.Display()' is called, it first prints "From Employee Class" from the overridden method, then uses 'super().Display()' to call the 'Display' method from the 'Person' class, which prints "From Person Class." This allows 'Employee' to extend the functionality of 'Person' while still using the parent class's method.

❖ __init__() function:

When the __init__() function is added in the child class, the child class will no longer inherit the parent's __init__() function. To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function.

Example:

```python
class Person():
  def __init__(self, name, id):
    self.name = name
    self.id = id

  def Display(self):
    print("Name:", self.name, "ID:", self.id)

class Employee(Person):
  def __init__(self, name, id, salary):
    super().__init__(name, id)
    self.salary = salary

  def Display(self):
    print("Name:", self.name, "ID:", self.id, "Salary:", self.salary)


emp = Employee("Dan", 123, 10000)
emp.Display()
emp.salary = 15000
emp.Display()


# output
'''
Name: Dan ID: 123 Age: Salary: 10000
Name: Dan ID: 123 Age: Salary: 15000
'''
```

The code demonstrates inheritance and method overriding, with 'Employee' as a subclass of 'Person'. The 'Person' class has an '__init__' method to initialize 'name' and 'id', and a 'Display' method to print them. The 'Employee' class overrides the '__init__' method to include 'salary' and uses 'super().__init__(name, id)' to call the parent class's constructor. It also overrides the 'Display' method to print 'name', 'id', and 'salary'. When an 'Employee' object 'emp' is created and its 'Display' method is called, it prints the initial salary. After updating 'emp.salary', calling 'Display' again reflects the updated salary in the output.

❖ Private Variables in Python:

In definition, private variables would be those that can only be seen and accessed by members of the class to which they belong, not by members of any other class. When the programme runs, these variables are utilized to access the values to keep the information secret from other classes. Even the object of that class cannot access this private variable directly. It is only accessible via any method. In Python, it is accomplished using two underscores before the name of the attributes or the method.

Example 1:

```
Python
class Person():
  def __init__(self, name, id):
    self.name = name
    self.__id = id      # making the variable private

  def Display(self):
    print("Name:", self.name, "ID:", self.__id)

  def changeID(self, id):
    self.__id = id


# creating a object and show details
p = Person('Nobita', 1)
p.Display()

# changing the value of id directly
p.__id = 2
# it will not throw any error but nothing will be changed
# as private attributes are not directly accessible even if by own object
p.Display()

# if you want to change any private variables, you have to use
# some method that can access that private variables.
# Because private variables are accessible inside any method
# of that class, but not any child/sub class.
p.changeID(3)
p.Display()

# the following line throws an error as private methods are accessible by
# methods inside the class but not accessible by object.
# p.__privateMethod()


# output
'''
Name: Nobita ID: 1
Name: Nobita ID: 1
Name: Nobita ID: 3
'''
```

Example 1:

Python
```
class Person():
  def __init__(self, name, id):
    self.name = name
    self.__id = id      # making the variable private

  def Display(self):
    print("Name:", self.name, "ID:", self.__id)

  def showID(self):
    return self.__id


class Employee(Person):
  def __init__(self, name, id, salary):
    super().__init__(name, id)
    self.salary = salary

  def Display(self):
    print("Name:", self.name)
    print("Salary:", self.salary)

    # following line throws an error as private variable is not accessible
    # print("ID:", self.__id)
    print("ID:", self.showID())
    # parent class method can be accessed by 'self.mehtodName()'
    # if no other method is present in the child class with same name


emp = Employee("Dan", 123, 10000)
emp.Display()

# output
'''
Name: Nobita ID: 1
Name: Nobita ID: 1
Name: Nobita ID: 3
'''
```

❖ Types of Inheritance

1. Single Inheritance: A child class inherits from a single parent class.

Syntax:

```python
class Parent:
    pass

class Child(Parent):
    pass
```

2. Multilevel Inheritance: A child class inherits from a parent class, which in turn inherits from another parent class.

Syntax:

```python
class Grandparent:
    pass

class Parent(Grandparent):
    pass

class Child(Parent):
    pass
```

3. Hierarchical Inheritance: Multiple child classes inherit from the same parent class.

Syntax:

```python
class Parent:
    pass

class Child1(Parent):
    pass

class Child2(Parent):
    pass
```

4. Multiple Inheritance: A child class inherits from more than one parent class.

Syntax:

```python
class Parent1:
    pass

class Parent2:
    pass
```

```
class Child(Parent1, Parent2):
    pass
```

5. Hybrid Inheritance: A combination of two or more types of inheritance.

Example:

```python
class Parent1:
    pass

class Parent2:
    pass

class Child(Parent1, Parent2):
    pass

class GrandChild1(Child):
    pass

class GrandChild2(Child):
    pass
```