

Prepared by Asif Bhat

Python Tutorial

```
In [103]: import sys
import keyword
import operator
from datetime import datetime
import os
```

Keywords

Keywords are the reserved words in Python and can't be used as an identifier

```
In [3]: print(keyword.kwlist) # List all Python Keywords
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif',
'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'p
ass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

```
In [4]: len(keyword.kwlist) # Python contains 35 keywords
```

```
Out[4]: 35
```

Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

```
In [13]: 1var = 10 # Identifier can't start with a digit

File "<ipython-input-13-37e58aaf2d3b>", line 1
    1var = 10 # Identifier can't start with a digit
      ^
SyntaxError: invalid syntax
```

```
In [14]: val2@ = 35 # Identifier can't use special symbols

File "<ipython-input-14-cfbf60736601>", line 1
    val2@ = 35 # Identifier can't use special symbols
      ^
SyntaxError: invalid syntax
```

```
In [15]: import = 125 # Keywords can't be used as identifiers

File "<ipython-input-15-f7061d4fc9ba>", line 1
    import = 125 # Keywords can't be used as identifiers
      ^
SyntaxError: invalid syntax
```

```
In [16]: """
Correct way of defining an identifier
(Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore)
"""

val2 = 10
```

```
In [17]: val_ = 99
```

Comments in Python

Comments can be used to explain the code for more readability.

```
In [18]: # Single line comment  
val1 = 10
```

```
In [19]: # Multiple  
# line  
# comment  
val1 = 10
```

```
In [20]: '''  
Multiple  
line  
comment  
'''  
val1 = 10
```

```
In [21]: """  
Multiple  
line  
comment  
"""  
val1 = 10
```

Statements

Instructions that a Python interpreter can execute.

```
In [27]: # Single line statement  
p1 = 10 + 20  
p1
```

```
Out[27]: 30
```

```
In [28]: # Single Line statement
p2 = ['a' , 'b' , 'c' , 'd']
p2
```

```
Out[28]: ['a', 'b', 'c', 'd']
```

```
In [26]: # Multiple Line statement
p1 = 20 + 30 \
      + 40 + 50 +\
      + 70 + 80
p1
```

```
Out[26]: 290
```

```
In [29]: # Multiple Line statement
p2 = ['a' ,
      'b' ,
      'c' ,
      'd'
      ]
p2
```

```
Out[29]: ['a', 'b', 'c', 'd']
```

Indentation

Indentation refers to the spaces at the beginning of a code line. It is very important as Python uses indentation to indicate a block of code. If the indentation is not correct we will end up with **IndentationError** error.

```
In [37]: p = 10
if p == 10:
    print ('P is equal to 10') # correct indentation
```

```
P is equal to 10
```

In [38]: *# if indentation is skipped we will encounter "IndentationError: expected an indented block"*

```
p = 10
if p == 10:
print ('P is equal to 10')
```

File "<ipython-input-38-d7879ffaae93>", line 3

```
print ('P is equal to 10')
```

^

IndentationError: expected an indented block

In [39]:

```
for i in range(0,5):
    print(i)                # correct indentation
```

```
0
1
2
3
4
```

In [43]: *# if indentation is skipped we will encounter "IndentationError: expected an indented block"*

```
for i in range(0,5):
print(i)
```

File "<ipython-input-43-4a6de03bf63e>", line 2

```
print(i)
```

^

IndentationError: expected an indented block

In [45]:

```
for i in range(0,5): print(i) # correct indentation but less readable
```

```
0
1
2
3
4
```

```
In [48]: j=20
         for i in range(0,5):
             print(i) # inside the for loop
         print(j) # outside the for loop
```

```
0
1
2
3
4
20
```

Docstrings

1) Docstrings provide a convenient way of associating documentation with functions, classes, methods or modules.

2) They appear right after the definition of a function, method, class, or module.

```
In [49]: def square(num):
         '''Square Function :- This function will return the square of a number'''
         return num**2
```

```
In [51]: square(2)
```

```
Out[51]: 4
```

```
In [52]: square.__doc__ # We can access the Docstring using __doc__ method
```

```
Out[52]: 'Square Function :- This function will return the square of a number'
```

```
In [53]: def evenodd(num):
         '''evenodd Function :- This function will test whether a numbr is Even or Odd'''
         if num % 2 == 0:
             print("Even Number")
         else:
             print("Odd Number")
```

```
In [54]: evenodd(3)
```

Odd Number

```
In [55]: evenodd(2)
```

Even Number

```
In [56]: evenodd.__doc__
```

```
Out[56]: 'evenodd Function :- This function will test whether a numbr is Even or Odd'
```

Variables

A Python variable is a reserved memory location to store values. A variable is created the moment you first assign a value to it.

```
In [75]: p = 30
```

```
In [76]: '''
id() function returns the "identity" of the object.
The identity of an object - Is an integer
                        - Guaranteed to be unique
                        - Constant for this object during its lifetime.
'''
id(p)
```

```
Out[76]: 140735029552432
```

```
In [77]: hex(id(p)) # Memory address of the variable
```

```
Out[77]: '0x7fff6d71a530'
```

```
In [94]: p = 20 #Creates an integer object with value 20 and assigns the variable p to point to that object.  
q = 20 # Create new reference q which will point to value 20. p & q will be pointing to same memory location.  
r = q # variable r will also point to the same location where p & q are pointing/  
p, type(p), hex(id(p)) # Variable P is pointing to memory location '0x7fff6d71a3f0' where value 20 is stored
```

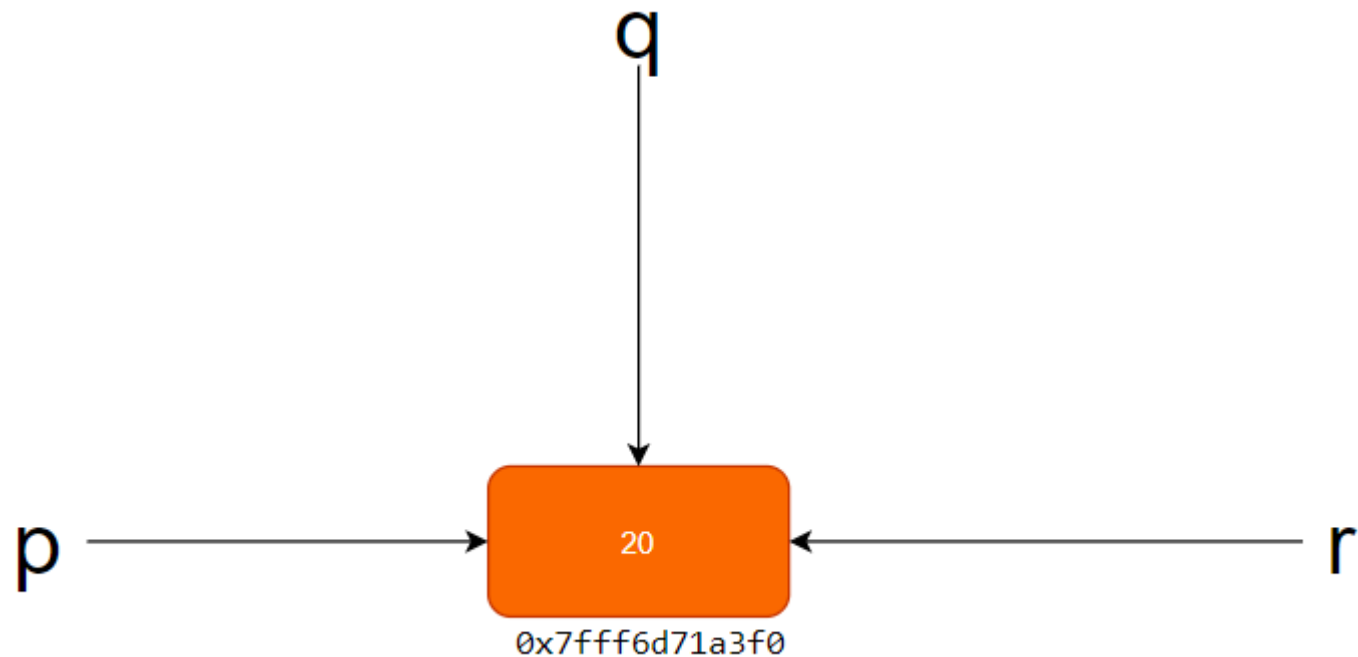
```
Out[94]: (20, int, '0x7fff6d71a3f0')
```

```
In [95]: q, type(q), hex(id(q))
```

```
Out[95]: (20, int, '0x7fff6d71a3f0')
```

```
In [96]: r, type(r), hex(id(r))
```

```
Out[96]: (20, int, '0x7fff6d71a3f0')
```




```
In [146]: p = 20  
p = p + 10 # Variable Overwriting  
p
```

Out[146]: 30

Variable Assignment

```
In [100]: intvar = 10 # Integer variable  
floatvar = 2.57 # Float Variable  
strvar = "Python Language" # String variable  
  
print(intvar)  
print(floatvar)  
print(strvar)
```

10
2.57
Python Language

Multiple Assignments

```
In [102]: intvar , floatvar , strvar = 10,2.57,"Python Language" # Using commas to separate variables and their corresponding values  
print(intvar)  
print(floatvar)  
print(strvar)
```

10
2.57
Python Language

```
In [105]: p1 = p2 = p3 = p4 = 44 # ALL variables pointing to same value  
print(p1,p2,p3,p4)
```

44 44 44 44

Data Types

Numeric

```
In [135]: val1 = 10 # Integer data type
print(val1)
print(type(val1)) # type of object
print(sys.getsizeof(val1)) # size of integer object in bytes
print(val1, " is Integer?", isinstance(val1, int)) # val1 is an instance of int class

10
<class 'int'>
28
10 is Integer? True
```

```
In [126]: val2 = 92.78 # Float data type
print(val2)
print(type(val2)) # type of object
print(sys.getsizeof(val2)) # size of float object in bytes
print(val2, " is float?", isinstance(val2, float)) # Val2 is an instance of float class

92.78
<class 'float'>
24
92.78 is float? True
```

```
In [136]: val3 = 25 + 10j # Complex data type
print(val3)
print(type(val3)) # type of object
print(sys.getsizeof(val3)) # size of float object in bytes
print(val3, " is complex?", isinstance(val3, complex)) # val3 is an instance of complex class

(25+10j)
<class 'complex'>
32
(25+10j) is complex? True
```

```
In [119]: sys.getsizeof(int()) # size of integer object in bytes
```

```
Out[119]: 24
```

```
In [120]: sys.getsizeof(float()) # size of float object in bytes
```

```
Out[120]: 24
```

```
In [138]: sys.getsizeof(complex()) # size of complex object in bytes
```

```
Out[138]: 32
```

Boolean

Boolean data type can have only two possible values **true** or **false**.

```
In [139]: bool1 = True
```

```
In [140]: bool2 = False
```

```
In [143]: print(type(bool1))
```

```
<class 'bool'>
```

```
In [144]: print(type(bool2))
```

```
<class 'bool'>
```

```
In [148]: isinstance(bool1, bool)
```

```
Out[148]: True
```

```
In [235]: bool(0)
```

```
Out[235]: False
```

```
In [236]: bool(1)
```

```
Out[236]: True
```

```
In [237]: bool(None)
```

```
Out[237]: False
```

```
In [238]: bool (False)
```

```
Out[238]: False
```

Strings

String Creation

```
In [193]: str1 = "HELLO PYTHON"  
  
print(str1)
```

```
HELLO PYTHON
```

```
In [194]: mystr = 'Hello World' # Define string using single quotes  
print(mystr)
```

```
Hello World
```

```
In [195]: mystr = "Hello World" # Define string using double quotes  
print(mystr)
```

```
Hello World
```

```
In [196]: mystr = '''Hello
           World ''' # Define string using triple quotes
print(mystr)
```

Hello
World

```
In [197]: mystr = """Hello
           World""" # Define string using triple quotes
print(mystr)
```

Hello
World

```
In [198]: mystr = ('Happy '
                   'Monday '
                   'Everyone')
print(mystr)
```

Happy Monday Everyone

```
In [199]: mystr2 = 'Woohoo '
mystr2 = mystr2*5
mystr2
```

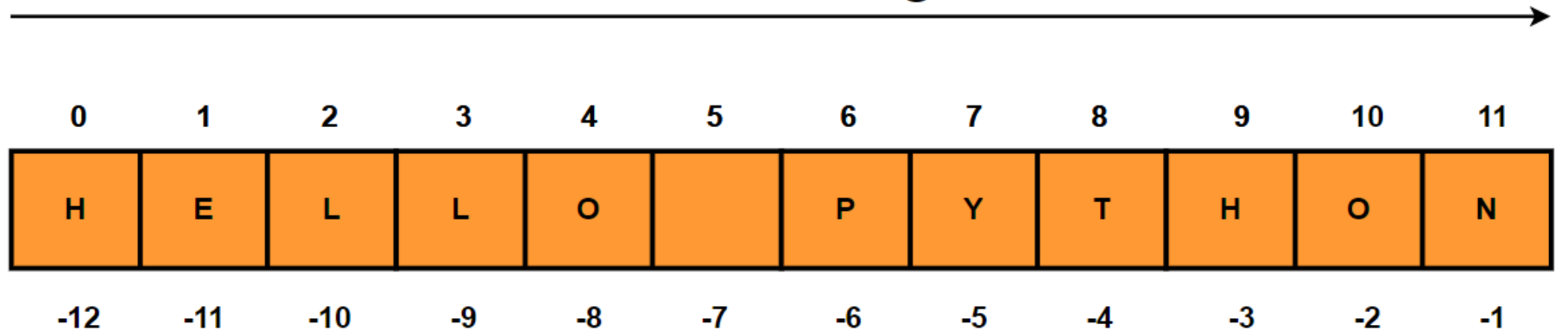
```
Out[199]: 'Woohoo Woohoo Woohoo Woohoo Woohoo '
```

```
In [200]: len(mystr2) # Length of string
```

```
Out[200]: 35
```

String Indexing

Forward Indexing



Backward Indexing

```
In [201]: str1
```

```
Out[201]: 'HELLO PYTHON'
```

```
In [202]: str1[0] # First character in string "str1"
```

```
Out[202]: 'H'
```

```
In [203]: str1[len(str1)-1] # Last character in string using len function
```

```
Out[203]: 'N'
```

```
In [204]: str1[-1] # Last character in string
```

```
Out[204]: 'N'
```

```
In [205]: str1[6] #Fetch 7th element of the string
```

```
Out[205]: 'P'
```

```
In [206]: str1[5]
```

```
Out[206]: ' '
```

String Slicing

```
In [207]: str1[0:5] # String slicing - Fetch all characters from 0 to 5 index location excluding the character at loc 5.
```

```
Out[207]: 'HELLO'
```

```
In [208]: str1[6:12] # String slicing - Retrieve all characters between 6 - 12 index loc excluding index loc 12.
```

```
Out[208]: 'PYTHON'
```

```
In [209]: str1[-4:] # Retrieve last four characters of the string
```

```
Out[209]: 'THON'
```

```
In [210]: str1[-6:] # Retrieve last six characters of the string
```

```
Out[210]: 'PYTHON'
```

```
In [211]: str1[:4] # Retrieve first four characters of the string
```

```
Out[211]: 'HELL'
```

```
In [212]: str1[:6] # Retrieve first six characters of the string
```

```
Out[212]: 'HELLO '
```

Update & Delete String

```
In [213]: str1
```

```
Out[213]: 'HELLO PYTHON'
```

```
In [214]: #Strings are immutable which means elements of a string cannot be changed once they have been assigned.  
str1[0:5] = 'HOLAA'
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-214-ea670ff3ec72> in <module>  
      1 #Strings are immutable which means elements of a string cannot be changed once they have been assigned.  
----> 2 str1[0:5] = 'HOLAA'  
  
TypeError: 'str' object does not support item assignment
```

```
In [215]: del str1 # Delete a string  
print(srt1)
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-215-7fcc0cc83dcc> in <module>  
      1 del str1 # Delete a string  
----> 2 print(srt1)  
  
NameError: name 'srt1' is not defined
```

String concatenation

```
In [216]: # String concatenation  
s1 = "Hello"  
s2 = "Asif"  
s3 = s1 + s2  
print(s3)
```

HelloAsif


```
In [217]: # String concatenation
s1 = "Hello"
s2 = "Asif"
s3 = s1 + " " + s2
print(s3)
```

Hello Asif

Iterating through a String

```
In [218]: mystr1 = "Hello Everyone"
```

```
In [219]: # Iteration
for i in mystr1:
    print(i)
```

H
e
l
l
o

E
v
e
r
y
o
n
e

```
In [220]: for i in enumerate(mystr1):  
          print(i)
```

```
(0, 'H')  
(1, 'e')  
(2, 'l')  
(3, 'l')  
(4, 'o')  
(5, ' ')  
(6, 'E')  
(7, 'v')  
(8, 'e')  
(9, 'r')  
(10, 'y')  
(11, 'o')  
(12, 'n')  
(13, 'e')
```

```
In [221]: list(enumerate(mystr1)) # Enumerate method adds a counter to an iterable and returns it in a form of enumerate object.
```

```
Out[221]: [(0, 'H'),  
          (1, 'e'),  
          (2, 'l'),  
          (3, 'l'),  
          (4, 'o'),  
          (5, ' '),  
          (6, 'E'),  
          (7, 'v'),  
          (8, 'e'),  
          (9, 'r'),  
          (10, 'y'),  
          (11, 'o'),  
          (12, 'n'),  
          (13, 'e')]
```

String Membership

In [222]: *# String membership*

```
mystr1 = "Hello Everyone"
```

```
print ('Hello' in mystr1) # Check whether substring "Hello" is present in string "mysrt1"
print ('Everyone' in mystr1) # Check whether substring "Everyone" is present in string "mysrt1"
print ('Hi' in mystr1) # Check whether substring "Hi" is present in string "mysrt1"
```

True

True

False

String Partitioning

In [256]:

```
"""
The partition() method searches for a specified string and splits the string into a tuple containing three elements.

- The first element contains the part before the argument string.

- The second element contains the argument string.

- The third element contains the part after the argument string.
"""
```

```
str5 = "Natural language processing with Python and R and Java"
L = str5.partition("and")
print(L)
```

```
('Natural language processing with Python ', 'and', ' R and Java')
```

```
In [257]: """
The rpartition() method searches for the last occurrence of the specified string and splits the string into a tuple
containing three elements.

- The first element contains the part before the argument string.

- The second element contains the argument string.

- The third element contains the part after the argument string.
"""

str5 = "Natural language processing with Python and R and Java"
L = str5.rpartition("and")
print(L)

('Natural language processing with Python and R ', 'and', ' Java')
```

String Functions

```
In [267]: mystr2 = "  Hello Everyone  "
mystr2
```

```
Out[267]: '  Hello Everyone  '
```

```
In [268]: mystr2.strip() # Removes white space from begining & end
```

```
Out[268]: 'Hello Everyone'
```

```
In [270]: mystr2.rstrip() # Removes all whitespaces at the end of the string
```

```
Out[270]: '  Hello Everyone'
```

```
In [269]: mystr2.lstrip() # Removes all whitespaces at the begining of the string
```

```
Out[269]: 'Hello Everyone  '
```

```
In [272]: mystr2 = "*****Hello Everyone*****All the Best*****"  
mystr2
```

```
Out[272]: '*****Hello Everyone*****All the Best*****'
```

```
In [273]: mystr2.strip('*') # Removes all '*' characters from begining & end of the string
```

```
Out[273]: 'Hello Everyone*****All the Best'
```

```
In [274]: mystr2.rstrip('*') # Removes all '*' characters at the end of the string
```

```
Out[274]: '*****Hello Everyone*****All the Best'
```

```
In [275]: mystr2.lstrip('*') # Removes all '*' characters at the begining of the string
```

```
Out[275]: 'Hello Everyone*****All the Best*****'
```

```
In [276]: mystr2 = "    Hello Everyone    "
```

```
In [277]: mystr2.lower() # Return whole string in lowercase
```

```
Out[277]: '    hello everyone    '
```

```
In [278]: mystr2.upper() # Return whole string in uppercase
```

```
Out[278]: '    HELLO EVERYONE    '
```

```
In [279]: mystr2.replace("He" , "Ho") #Replace substring "He" with "Ho"
```

```
Out[279]: '    Hollo Everyone    '
```

```
In [280]: mystr2.replace(" " , "") # Remove all whitespaces using replace function
```

```
Out[280]: 'HelloEveryone'
```

```
In [281]: mystr5 = "one two Three one two two three"
```

```
In [230]: mystr5.count("one") # Number of times substring "one" occurred in string.
```

```
Out[230]: 2
```

```
In [231]: mystr5.count("two") # Number of times substring "two" occurred in string.
```

```
Out[231]: 3
```

```
In [232]: mystr5.startswith("one") # Return boolean value True if string starts with "one"
```

```
Out[232]: True
```

```
In [233]: mystr5.endswith("three") # Return boolean value True if string ends with "three"
```

```
Out[233]: True
```

```
In [234]: mystr4 = "one two three four one two two three five five six seven six seven one one one ten eight ten nine eleven ten te
```

```
In [235]: mylist = mystr4.split() # Split String into substrings
mylist
```

```
Out[235]: ['one',
           'two',
           'three',
           'four',
           'one',
           'two',
           'two',
           'three',
           'five',
           'five',
           'six',
           'seven',
           'six',
           'seven',
           'one',
           'one',
           'one',
           'ten',
           'eight',
           'ten',
           'nine',
           'eleven',
           'ten',
           'ten',
           'nine']
```

```
In [236]: # Combining string & numbers using format method
item1 = 40
item2 = 55
item3 = 77

res = "Cost of item1 , item2 and item3 are {} , {} and {}"

print(res.format(item1,item2,item3))
```

Cost of item1 , item2 and item3 are 40 , 55 and 77

```
In [237]: # Combining string & numbers using format method
item1 = 40
item2 = 55
item3 = 77

res = "Cost of item3 , item2 and item1 are {2} , {1} and {0}"

print(res.format(item1,item2,item3))
```

Cost of item3 , item2 and item1 are 77 , 55 and 40

```
In [238]: str2 = " WELCOME EVERYONE "
str2 = str2.center(100) # center align the string using a specific character as the fill character.
print(str2)
```

WELCOME EVERYONE

```
In [239]: str2 = " WELCOME EVERYONE "
str2 = str2.center(100, '*') # center align the string using a specific character ('*') as the fill character.
print(str2)
```

***** WELCOME EVERYONE *****

```
In [240]: str2 = " WELCOME EVERYONE "
str2 = str2.rjust(50) # Right align the string using a specific character as the fill character.
print(str2)
```

WELCOME EVERYONE

```
In [241]: str2 = " WELCOME EVERYONE "
str2 = str2.rjust(50, '*') # Right align the string using a specific character ('*') as the fill character.
print(str2)
```

***** WELCOME EVERYONE


```
In [242]: str4 = "one two three four five six seven"
loc = str4.find("five") # Find the location of word 'five' in the string "str4"
print(loc)
```

19

```
In [243]: str4 = "one two three four five six seven"
loc = str4.index("five") # Find the location of word 'five' in the string "str4"
print(loc)
```

19

```
In [244]: mystr6 = '123456789'
print(mystr6.isalpha()) # returns True if all the characters in the text are letters
print(mystr6.isalnum()) # returns True if a string contains only letters or numbers or both
print(mystr6.isdecimal()) # returns True if all the characters are decimals (0-9)
print(mystr6.isnumeric()) # returns True if all the characters are numeric (0-9)
```

False

True

True

True

```
In [245]: mystr6 = 'abcde'
print(mystr6.isalpha()) # returns True if all the characters in the text are letters
print(mystr6.isalnum()) # returns True if a string contains only letters or numbers or both
print(mystr6.isdecimal()) # returns True if all the characters are decimals (0-9)
print(mystr6.isnumeric()) # returns True if all the characters are numeric (0-9)
```

True

True

False

False

```
In [246]: mystr6 = 'abc12309'
print(mystr6.isalpha()) # returns True if all the characters in the text are Letters
print(mystr6.isalnum()) # returns True if a string contains only Letters or numbers or both
print(mystr6.isdecimal()) # returns True if all the characters are decimals (0-9)
print(mystr6.isnumeric()) # returns True if all the characters are numeric (0-9)
```

```
False
True
False
False
```

```
In [247]: mystr7 = 'ABCDEF'
print(mystr7.isupper()) # Returns True if all the characters are in upper case
print(mystr7.islower()) # Returns True if all the characters are in lower case
```

```
True
False
```

```
In [248]: mystr8 = 'abcdef'
print(mystr8.isupper()) # Returns True if all the characters are in upper case
print(mystr8.islower()) # Returns True if all the characters are in lower case
```

```
False
True
```

```
In [258]: str6 = "one two three four one two two three five five six one ten eight ten nine eleven ten ten nine"

loc = str6.rfind("one") # Last occurrence of word 'one' in string "str6"
print(loc)
```

```
51
```

```
In [259]: loc = str6.rindex("one") # Last occurrence of word 'one' in string "str6"

print(loc)
```

```
51
```

```
In [264]: txt = "  abc def ghi  "
txt.rstrip()
```

```
Out[264]: '  abc def ghi'
```

```
In [265]: txt = "  abc def ghi  "
txt.lstrip()
```

```
Out[265]: 'abc def ghi  '
```

```
In [266]: txt = "  abc def ghi  "
txt.strip()
```

```
Out[266]: 'abc def ghi'
```

Using Escape Character

```
In [252]: #Using double quotes in the string is not allowed.
mystr = "My favourite TV Series is "Game of Thrones""
```

```
File "<ipython-input-252-0fa35a74da86>", line 2
    mystr = "My favourite TV Series is "Game of Thrones""
                                         ^
```

```
SyntaxError: invalid syntax
```

```
In [253]: #Using escape character to allow illegal characters
mystr = "My favourite series is \"Game of Thrones\""
print(mystr)
```

```
My favourite series is "Game of Thrones"
```

List

1) List is an ordered sequence of items.

2) We can have different data types under a list. E.g we can have integer, float and string items in a same list.

List Creation

```
In [423]: list1 = []      # Empty List
```

```
In [491]: print(type(list1))  
<class 'list'>
```

```
In [424]: list2 = [10,30,60]      # List of integers numbers
```

```
In [425]: list3 = [10.77,30.66,60.89]      # List of float numbers
```

```
In [426]: list4 = ['one','two' , "three"]  # List of strings
```

```
In [427]: list5 = ['Asif', 25 ,[50, 100],[150, 90]]      # Nested Lists
```

```
In [428]: list6 = [100, 'Asif', 17.765]      # List of mixed data types
```

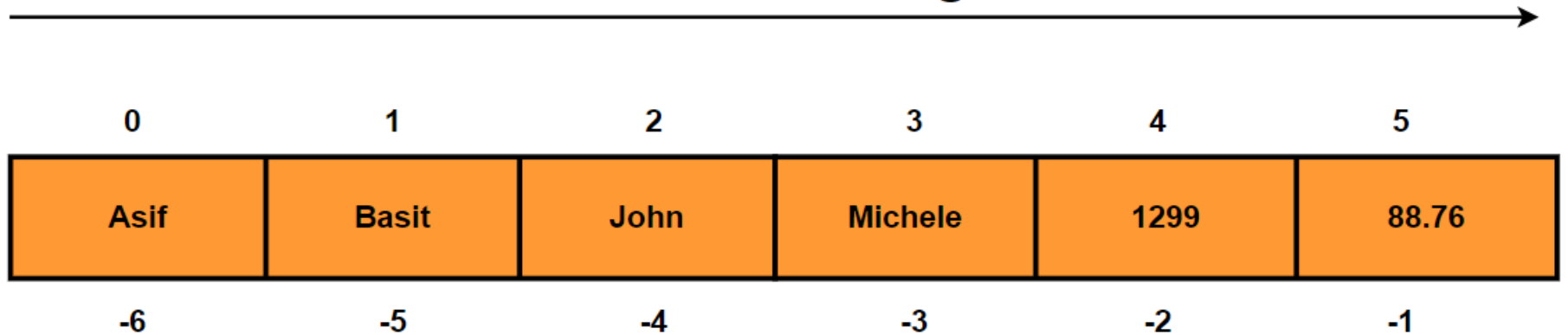
```
In [429]: list7 = ['Asif', 25 ,[50, 100],[150, 90] , {'John' , 'David'}]
```

```
In [430]: len(list6) #Length of list
```

```
Out[430]: 3
```

List Indexing

Forward Indexing



Backward Indexing

```
In [432]: list2[0] # Retrieve first element of the list
```

```
Out[432]: 10
```

```
In [433]: list4[0] # Retrieve first element of the list
```

```
Out[433]: 'one'
```

```
In [434]: list4[0][0] # Nested indexing - Access the first character of the first list element
```

```
Out[434]: 'o'
```

```
In [435]: list4[-1] # Last item of the list
```

```
Out[435]: 'three'
```

```
In [436]: list5[-1] # Last item of the list
```

```
Out[436]: [150, 90]
```

List Slicing

```
In [437]: mylist = ['one' , 'two' , 'three' , 'four' , 'five' , 'six' , 'seven' , 'eight']
```

```
In [438]: mylist[0:3] # Return all items from 0th to 3rd index location excluding the item at Loc 3.
```

```
Out[438]: ['one', 'two', 'three']
```

```
In [439]: mylist[2:5] # List all items from 2nd to 5th index location excluding the item at Loc 5.
```

```
Out[439]: ['three', 'four', 'five']
```

```
In [440]: mylist[:3] # Return first three items
```

```
Out[440]: ['one', 'two', 'three']
```

```
In [441]: mylist[:2] # Return first two items
```

```
Out[441]: ['one', 'two']
```

```
In [442]: mylist[-3:] # Return last three items
```

```
Out[442]: ['six', 'seven', 'eight']
```

```
In [443]: mylist[-2:] # Return last two items
```

```
Out[443]: ['seven', 'eight']
```

```
In [444]: mylist[-1] # Return last item of the list
```

```
Out[444]: 'eight'
```

```
In [445]: mylist[:] # Return whole list
```

```
Out[445]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

Add , Remove & Change Items

```
In [446]: mylist
```

```
Out[446]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [447]: mylist.append('nine') # Add an item to the end of the list  
mylist
```

```
Out[447]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine']
```

```
In [448]: mylist.insert(9, 'ten') # Add item at index location 9  
mylist
```

```
Out[448]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

```
In [449]: mylist.insert(1, 'ONE') # Add item at index location 1  
mylist
```

```
Out[449]: ['one',  
            'ONE',  
            'two',  
            'three',  
            'four',  
            'five',  
            'six',  
            'seven',  
            'eight',  
            'nine',  
            'ten']
```

```
In [450]: mylist.remove('ONE') # Remove item "ONE"  
mylist
```

```
Out[450]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

```
In [451]: mylist.pop() # Remove last item of the list  
mylist
```

```
Out[451]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine']
```

```
In [452]: mylist.pop(8) # Remove item at index location 8  
mylist
```

```
Out[452]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [453]: del mylist[7] # Remove item at index location 7  
mylist
```

```
Out[453]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven']
```

```
In [454]: # Change value of the string  
mylist[0] = 1  
mylist[1] = 2  
mylist[2] = 3  
mylist
```

```
Out[454]: [1, 2, 3, 'four', 'five', 'six', 'seven']
```

```
In [455]: mylist.clear() # Empty list / Delete all items in the list  
mylist
```

```
Out[455]: []
```



```
In [456]: del mylist # Delete the whole list
mylist
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-456-50c7849aa2cb> in <module>
      1 del mylist # Delete the whole list
----> 2 mylist

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

Copy List

```
In [457]: mylist = ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

```
In [458]: mylist1 = mylist # Create a new reference "mylist1"
```

```
In [459]: id(mylist) , id(mylist1) # The address of both mylist & mylist1 will be the same as both are pointing to same list.
```

```
Out[459]: (1537348392776, 1537348392776)
```

```
In [460]: mylist2 = mylist.copy() # Create a copy of the list
```

```
In [461]: id(mylist2) # The address of mylist2 will be different from mylist because mylist2 is pointing to the copy of the existing list
```

```
Out[461]: 1537345955016
```

```
In [462]: mylist[0] = 1
```

```
In [463]: mylist
```

```
Out[463]: [1, 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

```
In [464]: mylist1 # mylist1 will be also impacted as it is pointing to the same list
```

```
Out[464]: [1, 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

```
In [465]: mylist2 # Copy of list won't be impacted due to changes made on the original list
```

```
Out[465]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

Join Lists

```
In [466]: list1 = ['one', 'two', 'three', 'four']  
list2 = ['five', 'six', 'seven', 'eight']
```

```
In [467]: list3 = list1 + list2 # Join two lists by '+' operator  
list3
```

```
Out[467]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [468]: list1.extend(list2) #Append list2 with list1  
list1
```

```
Out[468]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

List Membership

```
In [469]: list1
```

```
Out[469]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [470]: 'one' in list1 # Check if 'one' exist in the list
```

```
Out[470]: True
```

```
In [471]: 'ten' in list1 # Check if 'ten' exist in the list
```

```
Out[471]: False
```

```
In [472]: if 'three' in list1: # Check if 'three' exist in the list
          print('Three is present in the list')
        else:
          print('Three is not present in the list')
```

Three is present in the list

```
In [473]: if 'eleven' in list1: # Check if 'eleven' exist in the list
          print('eleven is present in the list')
        else:
          print('eleven is not present in the list')
```

eleven is not present in the list

Reverse & Sort List

```
In [474]: list1
```

```
Out[474]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [475]: list1.reverse() # Reverse the list
          list1
```

```
Out[475]: ['eight', 'seven', 'six', 'five', 'four', 'three', 'two', 'one']
```

```
In [476]: list1 = list1[::-1] # Reverse the list
          list1
```

```
Out[476]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [477]: mylist3 = [9,5,2,99,12,88,34]
          mylist3.sort() # Sort list in ascending order
          mylist3
```

```
Out[477]: [2, 5, 9, 12, 34, 88, 99]
```

```
In [478]: mylist3 = [9,5,2,99,12,88,34]
mylist3.sort(reverse=True) # Sort list in descending order
mylist3
```

```
Out[478]: [99, 88, 34, 12, 9, 5, 2]
```

```
In [584]: mylist4 = [88,65,33,21,11,98]
sorted(mylist4) # Returns a new sorted list and doesn't change original list
```

```
Out[584]: [11, 21, 33, 65, 88, 98]
```

```
In [585]: mylist4
```

```
Out[585]: [88, 65, 33, 21, 11, 98]
```

Loop through a list

```
In [481]: list1
```

```
Out[481]: ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']
```

```
In [482]: for i in list1:
           print(i)
```

```
one
two
three
four
five
six
seven
eight
```

```
In [483]: for i in enumerate(list1):  
          print(i)
```

```
(0, 'one')  
(1, 'two')  
(2, 'three')  
(3, 'four')  
(4, 'five')  
(5, 'six')  
(6, 'seven')  
(7, 'eight')
```

Count

```
In [485]: list10 = ['one', 'two', 'three', 'four', 'one', 'one', 'two', 'three']
```

```
In [486]: list10.count('one') # Number of times item "one" occurred in the list.
```

```
Out[486]: 3
```

```
In [487]: list10.count('two') # Occurrence of item 'two' in the list
```

```
Out[487]: 2
```

```
In [489]: list10.count('four') #Occurrence of item 'four' in the list
```

```
Out[489]: 1
```

All / Any

The **all()** method returns:

- **True** - If all elements in a list are true
- **False** - If any element in a list is false

The **any()** function returns True if any element in the list is True. If not, any() returns False.

```
In [816]: L1 = [1,2,3,4,0]
```

```
In [817]: all(L1) # Will Return false as one value is false (Value 0)
```

```
Out[817]: False
```

```
In [818]: any(L1) # Will Return True as we have items in the list with True value
```

```
Out[818]: True
```

```
In [819]: L2 = [1,2,3,4,True,False]
```

```
In [820]: all(L2) # Returns false as one value is false
```

```
Out[820]: False
```

```
In [821]: any(L2) # Will Return True as we have items in the list with True value
```

```
Out[821]: True
```

```
In [822]: L3 = [1,2,3,True]
```

```
In [823]: all(L3) # Will return True as all items in the list are True
```

```
Out[823]: True
```

```
In [824]: any(L3) # Will Return True as we have items in the list with True value
```

```
Out[824]: True
```

List Comprehensions

- List Comprehensions provide an elegant way to create new lists.
- It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses.

```
In [287]: mystring = "WELCOME"  
mylist = [ i for i in mystring ] # Iterating through a string Using List Comprehension  
mylist
```

```
Out[287]: ['W', 'E', 'L', 'C', 'O', 'M', 'E']
```

```
In [289]: mylist1 = [ i for i in range(40) if i % 2 == 0 ] # Display all even numbers between 0 - 40 using List Comprehension  
mylist1
```

```
Out[289]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

```
In [290]: mylist2 = [ i for i in range(40) if i % 2 == 1 ] # Display all odd numbers between 0 - 40 using List Comprehension  
mylist2
```

```
Out[290]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
```

```
In [325]: mylist3 = [num**2 for num in range(10)] # calculate square of all numbers between 0 - 10 using List Comprehension  
mylist3
```

```
Out[325]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

[expression for item in list]

[num**2 for num in range(10)]

```
In [317]: # Multiple whole list by 10
list1 = [2,3,4,5,6,7,8]
list1 = [i*10 for i in list1]
list1
```

```
Out[317]: [20, 30, 40, 50, 60, 70, 80]
```

```
In [299]: #List all numbers divisible by 3 , 9 & 12 using nested "if" with List Comprehension
mylist4 = [i for i in range(200) if i % 3 == 0 if i % 9 == 0 if i % 12 == 0]
mylist4
```

```
Out[299]: [0, 36, 72, 108, 144, 180]
```

```
In [309]: # Odd even test
l1 = [print("{} is Even Number".format(i)) if i%2==0 else print("{} is odd number".format(i)) for i in range(10)]

0 is Even Number
1 is odd number
2 is Even Number
3 is odd number
4 is Even Number
5 is odd number
6 is Even Number
7 is odd number
8 is Even Number
9 is odd number
```

```
In [315]: # Extract numbers from a string
mystr = "One 1 two 2 three 3 four 4 five 5 six 6789"
numbers = [i for i in mystr if i.isdigit()]
numbers
```

```
Out[315]: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```


In [316]: *# Extract Letters from a string*

```
mystr = "One 1 two 2 three 3 four 4 five 5 six 6789"  
numbers = [i for i in mystr if i.isalpha()]  
numbers
```

Out[316]: ['O',
'n',
'e',
't',
'w',
'o',
't',
'h',
'r',
'e',
'e',
'f',
'o',
'u',
'r',
'f',
'i',
'v',
'e',
's',
'i',
'x']

Tuples

1. Tuple is similar to List except that the objects in tuple are immutable which means we cannot change the elements of a tuple once assigned.
2. When we do not want to change the data over time, tuple is a preferred data type.
3. Iterating over the elements of a tuple is faster compared to iterating over a list.

Tuple Creation

```
In [533]: tup1 = ()      # Empty tuple
```

```
In [534]: tup2 = (10,30,60)      # tuple of integers numbers
```

```
In [535]: tup3 = (10.77,30.66,60.89)      # tuple of float numbers
```

```
In [536]: tup4 = ('one','two' , "three")  # tuple of strings
```

```
In [537]: tup5 = ('Asif', 25 ,(50, 100),(150, 90))  # Nested tuples
```

```
In [538]: tup6 = (100, 'Asif', 17.765)  # Tuple of mixed data types
```

```
In [539]: tup7 = ('Asif', 25 ,[50, 100],[150, 90] , {'John' , 'David'} , (99,22,33))
```

```
In [540]: len(tup7) #Length of list
```

```
Out[540]: 6
```

Tuple Indexing

```
In [541]: tup2[0] # Retrieve first element of the tuple
```

```
Out[541]: 10
```

```
In [542]: tup4[0] # Retrieve first element of the tuple
```

```
Out[542]: 'one'
```

```
In [543]: tup4[0][0] # Nested indexing - Access the first character of the first tuple element
```

```
Out[543]: 'o'
```

```
In [544]: tup4[-1] # Last item of the tuple
```

```
Out[544]: 'three'
```

```
In [545]: tup5[-1] # Last item of the tuple
```

```
Out[545]: (150, 90)
```

Tuple Slicing

```
In [560]: mytuple = ('one' , 'two' , 'three' , 'four' , 'five' , 'six' , 'seven' , 'eight')
```

```
In [547]: mytuple[0:3] # Return all items from 0th to 3rd index location excluding the item at loc 3.
```

```
Out[547]: ('one', 'two', 'three')
```

```
In [548]: mytuple[2:5] # List all items from 2nd to 5th index location excluding the item at loc 5.
```

```
Out[548]: ('three', 'four', 'five')
```

```
In [549]: mytuple[:3] # Return first three items
```

```
Out[549]: ('one', 'two', 'three')
```

```
In [550]: mytuple[:2] # Return first two items
```

```
Out[550]: ('one', 'two')
```

```
In [551]: mytuple[-3:] # Return last three items
```

```
Out[551]: ('six', 'seven', 'eight')
```

```
In [552]: mytuple[-2:] # Return last two items
```

```
Out[552]: ('seven', 'eight')
```

```
In [553]: mytuple[-1] # Return last item of the tuple
```

```
Out[553]: 'eight'
```

```
In [554]: mytuple[:] # Return whole tuple
```

```
Out[554]: ('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight')
```

Remove & Change Items

```
In [555]: mytuple
```

```
Out[555]: ('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight')
```

```
In [556]: del mytuple[0] # Tuples are immutable which means we can't DELETE tuple items
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-556-667a276aa503> in <module>  
----> 1 del mytuple[0]
```

TypeError: 'tuple' object doesn't support item deletion

```
In [557]: mytuple[0] = 1 # Tuples are immutable which means we can't CHANGE tuple items
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-557-4cf492702bfd> in <module>  
----> 1 mytuple[0] = 1
```

TypeError: 'tuple' object does not support item assignment

```
In [561]: del mytuple # Deleting entire tuple object is possible
```

Loop through a tuple

```
In [570]: mytuple
```

```
Out[570]: ('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight')
```

```
In [571]: for i in mytuple:  
          print(i)
```

```
one  
two  
three  
four  
five  
six  
seven  
eight
```

```
In [572]: for i in enumerate(mytuple):  
          print(i)
```

```
(0, 'one')  
(1, 'two')  
(2, 'three')  
(3, 'four')  
(4, 'five')  
(5, 'six')  
(6, 'seven')  
(7, 'eight')
```

Count

```
In [573]: mytuple1 = ('one', 'two', 'three', 'four', 'one', 'one', 'two', 'three')
```

```
In [574]: mytuple1.count('one') # Number of times item "one" occurred in the tuple.
```

```
Out[574]: 3
```

```
In [575]: mytuple1.count('two') # Occurence of item 'two' in the tuple
```

```
Out[575]: 2
```

```
In [576]: mytuple1.count('four') #Occurence of item 'four' in the tuple
```

```
Out[576]: 1
```

Tuple Membership

```
In [577]: mytuple
```

```
Out[577]: ('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight')
```

```
In [578]: 'one' in mytuple # Check if 'one' exist in the list
```

```
Out[578]: True
```

```
In [579]: 'ten' in mytuple # Check if 'ten' exist in the list
```

```
Out[579]: False
```

```
In [581]: if 'three' in mytuple: # Check if 'three' exist in the list
          print('Three is present in the tuple')
          else:
          print('Three is not present in the tuple')
```

Three is present in the tuple

```
In [583]: if 'eleven' in mytuple: # Check if 'eleven' exist in the list
          print('eleven is present in the tuple')
          else:
          print('eleven is not present in the tuple')
```

eleven is not present in the tuple

Index Position

```
In [586]: mytuple
```

```
Out[586]: ('one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight')
```

```
In [587]: mytuple.index('one') # Index of first element equal to 'one'
```

```
Out[587]: 0
```

```
In [590]: mytuple.index('five') # Index of first element equal to 'five'
```

```
Out[590]: 4
```

```
In [591]: mytuple1
```

```
Out[591]: ('one', 'two', 'three', 'four', 'one', 'one', 'two', 'three')
```

```
In [593]: mytuple1.index('one') # Index of first element equal to 'one'
```

```
Out[593]: 0
```

Sorting

```
In [594]: mytuple2 = (43,67,99,12,6,90,67)
```

```
In [595]: sorted(mytuple2) # Returns a new sorted list and doesn't change original tuple
```

```
Out[595]: [6, 12, 43, 67, 67, 90, 99]
```

```
In [596]: sorted(mytuple2, reverse=True) # Sort in descending order
```

```
Out[596]: [99, 90, 67, 67, 43, 12, 6]
```

Sets

1) Unordered & Unindexed collection of items.

- 2) Set elements are unique. Duplicate elements are not allowed.
- 3) Set elements are immutable (cannot be changed).
- 4) Set itself is mutable. We can add or remove items from it.

Set Creation

```
In [634]: myset = {1,2,3,4,5} # Set of numbers  
myset
```

```
Out[634]: {1, 2, 3, 4, 5}
```

```
In [635]: len(myset) #Length of the set
```

```
Out[635]: 5
```

```
In [636]: my_set = {1,1,2,2,3,4,5,5}  
my_set           # Duplicate elements are not allowed.
```

```
Out[636]: {1, 2, 3, 4, 5}
```

```
In [637]: myset1 = {1.79,2.08,3.99,4.56,5.45} # Set of float numbers  
myset1
```

```
Out[637]: {1.79, 2.08, 3.99, 4.56, 5.45}
```

```
In [638]: myset2 = {'Asif' , 'John' , 'Tyrion'} # Set of Strings  
myset2
```

```
Out[638]: {'Asif', 'John', 'Tyrion'}
```

```
In [639]: myset3 = {10,20, "Hola", (11, 22, 32)} # Mixed datatypes  
myset3
```

```
Out[639]: {(11, 22, 32), 10, 20, 'Hola'}
```



```
In [640]: myset3 = {10,20, "Hola", [11, 22, 32]} # set doesn't allow mutable items like lists
myset3
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-640-d23fdc3a319e> in <module>
----> 1 myset3 = {10,20, "Hola", [11, 22, 32]} # set doesn't allow mutable items like lists
      2 myset3

TypeError: unhashable type: 'list'
```

```
In [641]: myset4 = set() # Create an empty set
print(type(myset4))
```

```
<class 'set'>
```

```
In [673]: my_set1 = set(('one' , 'two' , 'three' , 'four'))
my_set1
```

```
Out[673]: {'four', 'one', 'three', 'two'}
```

Loop through a Set

```
In [776]: myset = {'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight'}

for i in myset:
    print(i)
```

```
eight
one
seven
three
five
two
six
four
```

```
In [777]: for i in enumerate(myset):  
          print(i)
```

```
(0, 'eight')  
(1, 'one')  
(2, 'seven')  
(3, 'three')  
(4, 'five')  
(5, 'two')  
(6, 'six')  
(7, 'four')
```

Set Membership

```
In [675]: myset
```

```
Out[675]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [676]: 'one' in myset # Check if 'one' exist in the set
```

```
Out[676]: True
```

```
In [677]: 'ten' in myset # Check if 'ten' exist in the set
```

```
Out[677]: False
```

```
In [678]: if 'three' in myset: # Check if 'three' exist in the set  
          print('Three is present in the set')  
          else:  
          print('Three is not present in the set')
```

```
Three is present in the set
```

```
In [679]: if 'eleven' in myset: # Check if 'eleven' exist in the list
          print('eleven is present in the set')
          else:
          print('eleven is not present in the set')
```

eleven is not present in the set

Add & Remove Items

```
In [680]: myset
```

```
Out[680]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [681]: myset.add('NINE') # Add item to a set using add() method
          myset
```

```
Out[681]: {'NINE', 'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [683]: myset.update(['TEN' , 'ELEVEN' , 'TWELVE']) # Add multiple item to a set using update() method
          myset
```

```
Out[683]: {'ELEVEN',
          'NINE',
          'TEN',
          'TWELVE',
          'eight',
          'five',
          'four',
          'one',
          'seven',
          'six',
          'three',
          'two'}
```

```
In [684]: myset.remove('NINE') # remove item in a set using remove() method  
myset
```

```
Out[684]: {'ELEVEN',  
          'TEN',  
          'TWELVE',  
          'eight',  
          'five',  
          'four',  
          'one',  
          'seven',  
          'six',  
          'three',  
          'two'}
```

```
In [685]: myset.discard('TEN') # remove item from a set using discard() method  
myset
```

```
Out[685]: {'ELEVEN',  
          'TWELVE',  
          'eight',  
          'five',  
          'four',  
          'one',  
          'seven',  
          'six',  
          'three',  
          'two'}
```

```
In [688]: myset.clear() # Delete all items in a set  
myset
```

```
Out[688]: set()
```

```
In [689]: del myset # Delete the set object
myset
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-689-0912ea1b8932> in <module>
      1 del myset
----> 2 myset

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

Copy Set

```
In [705]: myset = {'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight'}
myset
```

```
Out[705]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [706]: myset1 = myset # Create a new reference "myset1"
myset1
```

```
Out[706]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [707]: id(myset) , id(myset1) # The address of both myset & myset1 will be the same as both are pointing to same set.
```

```
Out[707]: (1537349033320, 1537349033320)
```

```
In [708]: my_set = myset.copy() # Create a copy of the list
my_set
```

```
Out[708]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [710]: id(my_set) # The address of my_set will be different from myset because my_set is pointing to the copy of the existing set
```

```
Out[710]: 1537352902024
```

```
In [711]: myset.add('nine')  
myset
```

```
Out[711]: {'eight', 'five', 'four', 'nine', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [712]: myset1 # myset1 will be also impacted as it is pointing to the same Set
```

```
Out[712]: {'eight', 'five', 'four', 'nine', 'one', 'seven', 'six', 'three', 'two'}
```

```
In [713]: my_set # Copy of the set won't be impacted due to changes made on the original Set.
```

```
Out[713]: {'eight', 'five', 'four', 'one', 'seven', 'six', 'three', 'two'}
```

Set Operation

Union

```
In [757]: A = {1,2,3,4,5}  
B = {4,5,6,7,8}  
C = {8,9,10}
```

```
In [758]: A | B # Union of A and B (ALL elements from both sets. NO DUPLICATES)
```

```
Out[758]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [759]: A.union(B) # Union of A and B
```

```
Out[759]: {1, 2, 3, 4, 5, 6, 7, 8}
```

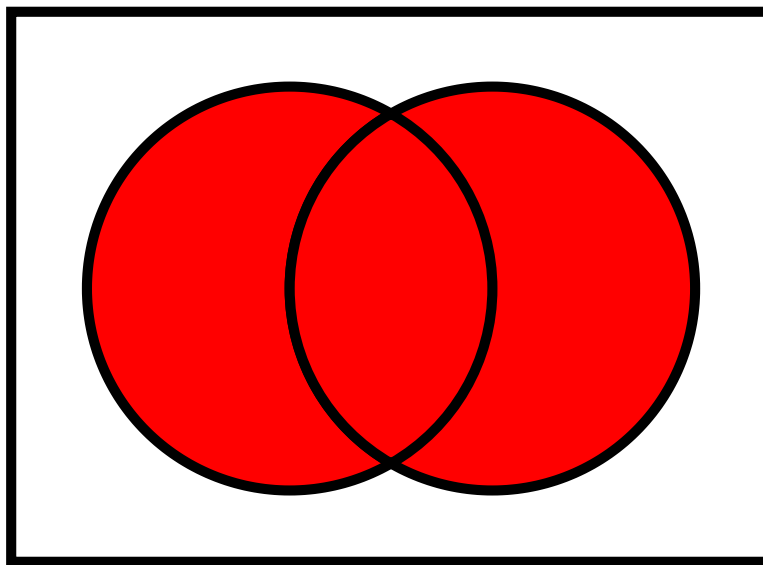
```
In [760]: A.union(B, C) # Union of A, B and C.
```

```
Out[760]: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
In [761]: """
Updates the set calling the update() method with union of A , B & C.

For below example Set A will be updated with union of A,B & C.
"""
A.update(B,C)
A
```

```
Out[761]: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```



Intersection

```
In [762]: A = {1,2,3,4,5}
          B = {4,5,6,7,8}
```

```
In [763]: A & B  # Intersection of A and B (Common items in both sets)
```

```
Out[763]: {4, 5}
```

```
In [764]: A.intersection(B)  Intersection of A and B
```

```
File "<ipython-input-764-f01b60f4d31d>", line 1
```

```
A.intersection(B)  Intersection of A and B
```

^

```
SyntaxError: invalid syntax
```

```
In [765]: """
Updates the set calling the intersection_update() method with the intersection of sets.
```

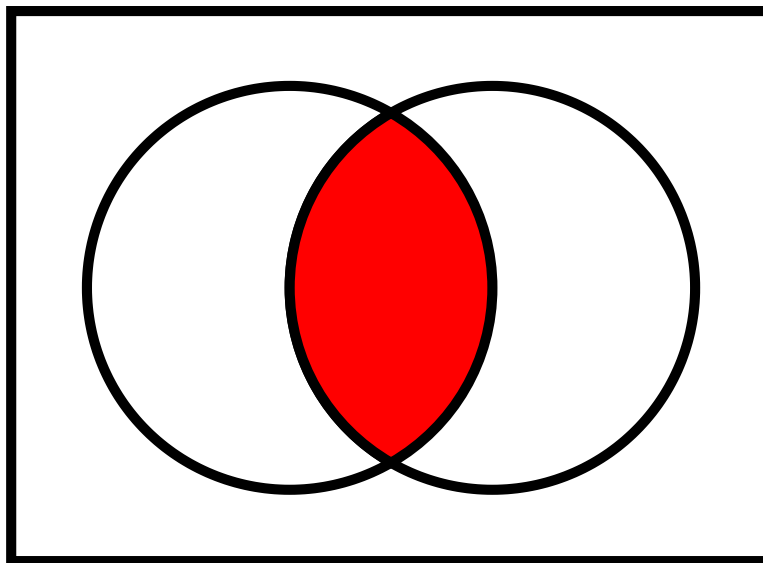
```
For below example Set A will be updated with the intersection of A & B.
```

```
"""
```

```
A.intersection_update(B)
```

```
A
```

```
Out[765]: {4, 5}
```



Difference


```
In [766]: A = {1,2,3,4,5}
          B = {4,5,6,7,8}
```

```
In [767]: A - B  # set of elements that are only in A but not in B
```

```
Out[767]: {1, 2, 3}
```

```
In [768]: A.difference(B)  # Difference of sets
```

```
Out[768]: {1, 2, 3}
```

```
In [769]: B - A  # set of elements that are only in B but not in A
```

```
Out[769]: {6, 7, 8}
```

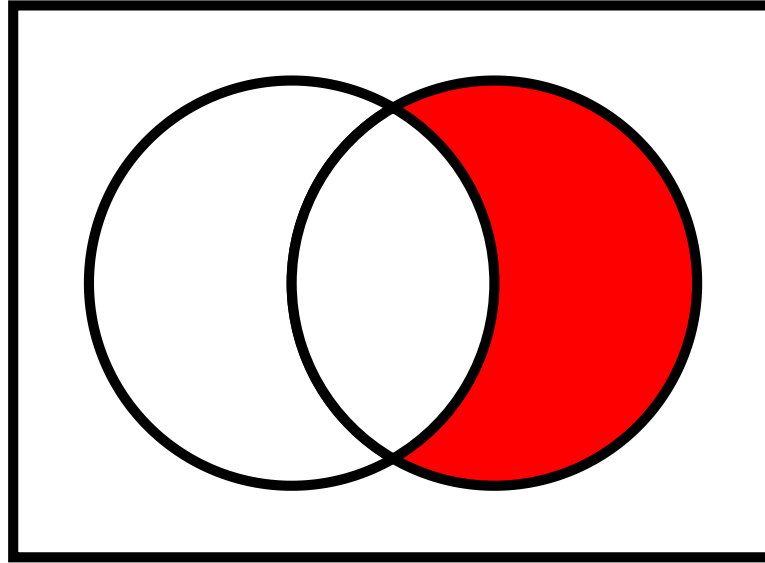
```
In [770]: B.difference(A)
```

```
Out[770]: {6, 7, 8}
```

```
In [771]: """
          Updates the set calling the difference_update() method with the difference of sets.

          For below example Set B will be updated with the difference of B & A.
          """
          B.difference_update(A)
          B
```

```
Out[771]: {6, 7, 8}
```



Symmetric Difference

```
In [772]: A = {1,2,3,4,5}
          B = {4,5,6,7,8}
```

```
In [773]: A ^ B # Symmetric difference (Set of elements in A and B but not in both. "EXCLUDING THE INTERSECTION").
```

```
Out[773]: {1, 2, 3, 6, 7, 8}
```

```
In [774]: A.symmetric_difference(B) # Symmetric difference of sets
```

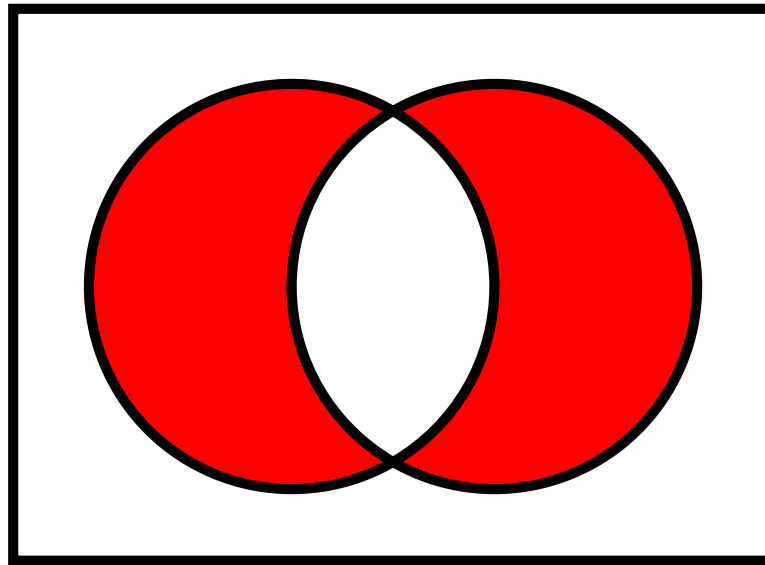
```
Out[774]: {1, 2, 3, 6, 7, 8}
```

```
In [775]: """
Updates the set calling the symmetric_difference_update() method with the symmetric difference of sets.

For below example Set A will be updated with the symmetric difference of A & B.
"""

A.symmetric_difference_update(B)
A
```

```
Out[775]: {1, 2, 3, 6, 7, 8}
```



Subset , Superset & Disjoint

```
In [784]: A = {1,2,3,4,5,6,7,8,9}
B = {3,4,5,6,7,8}
C = {10,20,30,40}
```

```
In [785]: B.issubset(A) # Set B is said to be the subset of set A if all elements of B are in A.
```

```
Out[785]: True
```

```
In [786]: A.issuperset(B) # Set A is said to be the superset of set B if all elements of B are in A.
```

```
Out[786]: True
```

```
In [787]: C.isdisjoint(A) # Two sets are said to be disjoint sets if they have no common elements.
```

```
Out[787]: True
```

```
In [788]: B.isdisjoint(A) # Two sets are said to be disjoint sets if they have no common elements.
```

```
Out[788]: False
```

Other Builtin functions

```
In [789]: A
```

```
Out[789]: {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
In [790]: sum(A)
```

```
Out[790]: 45
```

```
In [791]: max(A)
```

```
Out[791]: 9
```

```
In [792]: min(A)
```

```
Out[792]: 1
```

```
In [793]: len(A)
```

```
Out[793]: 9
```

```
In [795]: list(enumerate(A))
```

```
Out[795]: [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8), (8, 9)]
```

```
In [798]: D= sorted(A,reverse=True)  
D
```

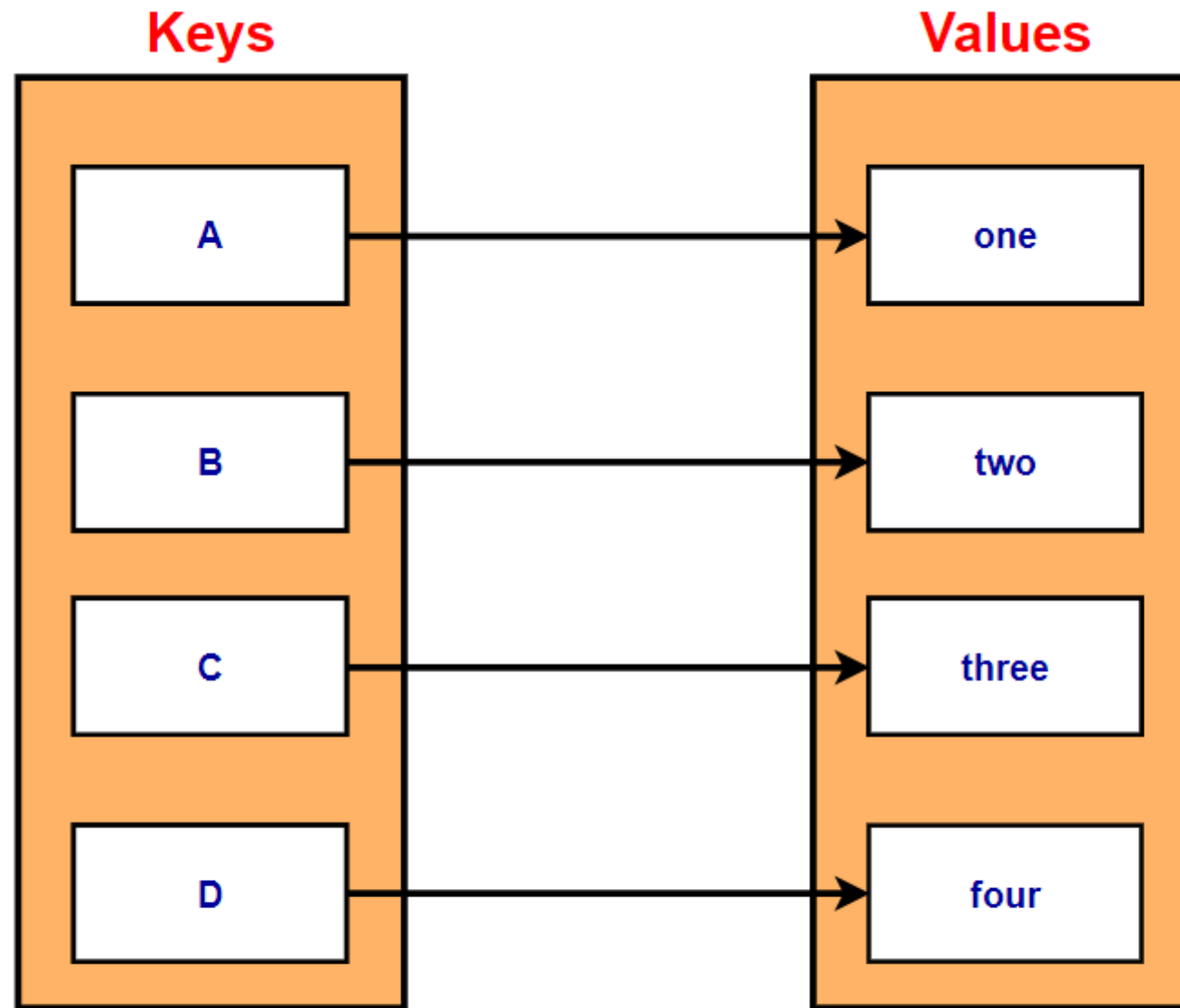
```
Out[798]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [799]: sorted(D)
```

```
Out[799]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dictionary

- Dictionary is a mutable data type in Python.
- A python dictionary is a collection of key and value pairs separated by a colon (:) & enclosed in curly braces {}.
- Keys must be unique in a dictionary, duplicate values are allowed.



```
mydict = {'A':'one' , 'B':'two' , 'C':'three' , 'D' : 'four'}
```

Create Dictionary

```
In [947]: mydict = dict() # empty dictionary  
mydict
```

```
Out[947]: {}
```

```
In [948]: mydict = {} # empty dictionary  
mydict
```

```
Out[948]: {}
```

```
In [949]: mydict = {1:'one' , 2:'two' , 3:'three'} # dictionary with integer keys  
mydict
```

```
Out[949]: {1: 'one', 2: 'two', 3: 'three'}
```

```
In [950]: mydict = dict({1:'one' , 2:'two' , 3:'three'}) # Create dictionary using dict()  
mydict
```

```
Out[950]: {1: 'one', 2: 'two', 3: 'three'}
```

```
In [951]: mydict = {'A':'one' , 'B':'two' , 'C':'three'} # dictionary with character keys  
mydict
```

```
Out[951]: {'A': 'one', 'B': 'two', 'C': 'three'}
```

```
In [318]: mydict = {1:'one' , 'A':'two' , 3:'three'} # dictionary with mixed keys  
mydict
```

```
Out[318]: {1: 'one', 'A': 'two', 3: 'three'}
```

```
In [319]: mydict.keys() # Return Dictionary Keys using keys() method
```

```
Out[319]: dict_keys([1, 'A', 3])
```

```
In [320]: mydict.values() # Return Dictionary Values using values() method
```

```
Out[320]: dict_values(['one', 'two', 'three'])
```

```
In [321]: mydict.items() # Access each key-value pair within a dictionary
```

```
Out[321]: dict_items([(1, 'one'), ('A', 'two'), (3, 'three')])
```

```
In [955]: mydict = {1:'one' , 2:'two' , 'A':['asif' , 'john' , 'Maria']} # dictionary with mixed keys  
mydict
```

```
Out[955]: {1: 'one', 2: 'two', 'A': ['asif', 'john', 'Maria']}
```

```
In [956]: mydict = {1:'one' , 2:'two' , 'A':['asif' , 'john' , 'Maria'], 'B':('Bat' , 'cat' , 'hat')} # dictionary with mixed keys  
mydict
```

```
Out[956]: {1: 'one',  
          2: 'two',  
          'A': ['asif', 'john', 'Maria'],  
          'B': ('Bat', 'cat', 'hat')}
```

```
In [1]: mydict = {1:'one' , 2:'two' , 'A':{'Name':'asif' , 'Age' :20}, 'B':('Bat' , 'cat' , 'hat')} # dictionary with mixed keys  
mydict
```

```
Out[1]: {1: 'one',  
        2: 'two',  
        'A': {'Name': 'asif', 'Age': 20},  
        'B': ('Bat', 'cat', 'hat')}
```

```
In [957]: keys = {'a' , 'b' , 'c' , 'd'}  
mydict3 = dict.fromkeys(keys) # Create a dictionary from a sequence of keys  
mydict3
```

```
Out[957]: {'c': None, 'd': None, 'a': None, 'b': None}
```



```
In [958]: keys = {'a' , 'b' , 'c' , 'd'}  
value = 10  
mydict3 = dict.fromkeys(keys , value) # Create a dictionary from a sequence of keys with value  
mydict3
```

```
Out[958]: {'c': 10, 'd': 10, 'a': 10, 'b': 10}
```

```
In [959]: keys = {'a' , 'b' , 'c' , 'd'}  
value = [10,20,30]  
mydict3 = dict.fromkeys(keys , value) # Create a dictionary from a sequence of keys with value list  
mydict3
```

```
Out[959]: {'c': [10, 20, 30], 'd': [10, 20, 30], 'a': [10, 20, 30], 'b': [10, 20, 30]}
```

```
In [960]: value.append(40)  
mydict3
```

```
Out[960]: {'c': [10, 20, 30, 40],  
          'd': [10, 20, 30, 40],  
          'a': [10, 20, 30, 40],  
          'b': [10, 20, 30, 40]}
```

Accessing Items

```
In [961]: mydict = {1:'one' , 2:'two' , 3:'three' , 4:'four'}  
mydict
```

```
Out[961]: {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

```
In [962]: mydict[1] # Access item using key
```

```
Out[962]: 'one'
```

```
In [963]: mydict.get(1) # Access item using get() method
```

```
Out[963]: 'one'
```

```
In [964]: mydict1 = {'Name':'Asif' , 'ID': 74123 , 'DOB': 1991 , 'job' : 'Analyst'}  
mydict1
```

```
Out[964]: {'Name': 'Asif', 'ID': 74123, 'DOB': 1991, 'job': 'Analyst'}
```

```
In [965]: mydict1['Name']  # Access item using key
```

```
Out[965]: 'Asif'
```

```
In [966]: mydict1.get('job')  # Access item using get() method
```

```
Out[966]: 'Analyst'
```

Add, Remove & Change Items

```
In [967]: mydict1 = {'Name':'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Address' : 'Hilsinki'}  
mydict1
```

```
Out[967]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Hilsinki'}
```

```
In [968]: mydict1['DOB'] = 1992  # Changing Dictionary Items  
mydict1['Address'] = 'Delhi'  
mydict1
```

```
Out[968]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1992, 'Address': 'Delhi'}
```

```
In [969]: dict1 = {'DOB':1995}  
mydict1.update(dict1)  
mydict1
```

```
Out[969]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1995, 'Address': 'Delhi'}
```

```
In [970]: mydict1['Job'] = 'Analyst' # Adding items in the dictionary  
mydict1
```

```
Out[970]: {'Name': 'Asif',  
          'ID': 12345,  
          'DOB': 1995,  
          'Address': 'Delhi',  
          'Job': 'Analyst'}
```

```
In [971]: mydict1.pop('Job') # Removing items in the dictionary using Pop method  
mydict1
```

```
Out[971]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1995, 'Address': 'Delhi'}
```

```
In [972]: mydict1.popitem() # A random item is removed
```

```
Out[972]: ('Address', 'Delhi')
```

```
In [973]: mydict1
```

```
Out[973]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1995}
```

```
In [974]: del[mydict1['ID']] # Removing item using del method  
mydict1
```

```
Out[974]: {'Name': 'Asif', 'DOB': 1995}
```

```
In [975]: mydict1.clear() # Delete all items of the dictionary using clear method  
mydict1
```

```
Out[975]: {}
```

```
In [976]: del mydict1 # Delete the dictionary object
mydict1
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-976-da2fba4eca0f> in <module>
      1 del mydict1 # Delete the dictionary object
----> 2 mydict1

NameError: name 'mydict1' is not defined
```

Copy Dictionary

```
In [977]: mydict = {'Name': 'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Address' : 'Hilsinki'}
mydict
```

```
Out[977]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Hilsinki'}
```

```
In [978]: mydict1 = mydict # Create a new reference "mydict1"
```

```
In [979]: id(mydict) , id(mydict1) # The address of both mydict & mydict1 will be the same as both are pointing to same dictionary
```

```
Out[979]: (1537346312776, 1537346312776)
```

```
In [980]: mydict2 = mydict.copy() # Create a copy of the dictionary
```

```
In [981]: id(mydict2) # The address of mydict2 will be different from mydict because mydict2 is pointing to the copy of the existing dictionary
```

```
Out[981]: 1537345875784
```

```
In [982]: mydict['Address'] = 'Mumbai'
```

```
In [983]: mydict
```

```
Out[983]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Mumbai'}
```

```
In [984]: mydict1 # mydict1 will be also impacted as it is pointing to the same dictionary
```

```
Out[984]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Mumbai'}
```

```
In [985]: mydict2 # Copy of list won't be impacted due to the changes made in the original dictionary
```

```
Out[985]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Helsinki'}
```

Loop through a Dictionary

```
In [986]: mydict1 = {'Name': 'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Address' : 'Helsinki' , 'Job': 'Analyst'}  
mydict1
```

```
Out[986]: {'Name': 'Asif',  
          'ID': 12345,  
          'DOB': 1991,  
          'Address': 'Helsinki',  
          'Job': 'Analyst'}
```

```
In [987]: for i in mydict1:  
          print(i , ':' , mydict1[i]) # Key & value pair
```

```
Name : Asif  
ID : 12345  
DOB : 1991  
Address : Helsinki  
Job : Analyst
```

```
In [988]: for i in mydict1:  
          print(mydict1[i]) # Dictionary items
```

```
Asif  
12345  
1991  
Helsinki  
Analyst
```

Dictionary Membership

```
In [989]: mydict1 = {'Name': 'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Job': 'Analyst'}  
mydict1
```

```
Out[989]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Job': 'Analyst'}
```

```
In [990]: 'Name' in mydict1 # Test if a key is in a dictionary or not.
```

```
Out[990]: True
```

```
In [991]: 'Asif' in mydict1 # Membership test can be only done for keys.
```

```
Out[991]: False
```

```
In [992]: 'ID' in mydict1
```

```
Out[992]: True
```

```
In [993]: 'Address' in mydict1
```

```
Out[993]: False
```

All / Any

The **all()** method returns:

- **True** - If all all keys of the dictionary are true
- **False** - If any key of the dictionary is false

The **any()** function returns True if any key of the dictionary is True. If not, any() returns False.

```
In [995]: mydict1 = {'Name': 'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Job': 'Analyst'}  
mydict1
```

```
Out[995]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Job': 'Analyst'}
```

```
In [996]: all(mydict1) # Will Return false as one value is false (Value 0)
```

```
Out[996]: True
```

```
In [997]: any(mydict1) # Will Return True as we have items in the dictionary with True value
```

```
Out[997]: True
```

```
In [998]: mydict1[0] = 'test1'  
mydict1
```

```
Out[998]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Job': 'Analyst', 0: 'test1'}
```

```
In [999]: all(mydict1) # Returns false as one value is false
```

```
Out[999]: False
```

```
In [1000]: any(mydict1) # Will Return True as we have items in the dictionary with True value
```

```
Out[1000]: True
```

Dictionary Comprehension

```
In [323]: double = {i:i*2 for i in range(10)} #double each value using dict comprehension  
double
```

```
Out[323]: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}
```

```
In [327]: square = {i:i**2 for i in range(10)}  
square
```

```
Out[327]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

`{key:value for var in iterable}`

`{i : i**2 for i in range(10)}`

```
In [329]: key = ['one' , 'two' , 'three' , 'four' , 'five']  
         value = [1,2,3,4,5]  
  
         mydict = {k:v for (k,v) in zip(key,value)} # using dict comprehension to create dictionary  
         mydict
```

```
Out[329]: {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
```

```
In [332]: mydict1 = {'a':10 , 'b':20 , 'c':30 , 'd':40 , 'e':50}  
         mydict1 = {k:v/10 for (k,v) in mydict1.items()} # Divide all values in a dictionary by 10  
         mydict1
```

```
Out[332]: {'a': 1.0, 'b': 2.0, 'c': 3.0, 'd': 4.0, 'e': 5.0}
```



```
In [334]: str1 = "Natural Language Processing"

mydict2 = {k:v for (k,v) in enumerate(str1)} # Store enumerated values in a dictionary
mydict2
```

```
Out[334]: {0: 'N',
1: 'a',
2: 't',
3: 'u',
4: 'r',
5: 'a',
6: 'l',
7: ' ',
8: 'L',
9: 'a',
10: 'n',
11: 'g',
12: 'u',
13: 'a',
14: 'g',
15: 'e',
16: ' ',
17: 'P',
18: 'r',
19: 'o',
20: 'c',
21: 'e',
22: 's',
23: 's',
24: 'i',
25: 'n',
26: 'g'}
```

```
In [337]: str1 = "abcdefghijklmnopqrstuvwxyz"
mydict3 = {i:i.upper() for i in str1} # Lower to Upper Case
mydict3
```

```
Out[337]: {'a': 'A',
'b': 'B',
'c': 'C',
'd': 'D',
'e': 'E',
'f': 'F',
'g': 'G',
'h': 'H',
'i': 'I',
'j': 'J',
'k': 'K',
'l': 'L',
'm': 'M',
'n': 'N',
'o': 'O',
'p': 'P',
'q': 'Q',
'r': 'R',
's': 'S',
't': 'T',
'u': 'U',
'v': 'V',
'w': 'W',
'x': 'X',
'y': 'Y',
'z': 'Z'}
```

Word Frequency using dictionary

```
In [61]: mystr4 = "one two three four one two two three five five six seven six seven one one one ten eight ten nine eleven ten te
```

```
In [64]: mylist = mystr4.split() # Split String into substrings  
mylist
```

```
Out[64]: ['one',  
          'two',  
          'three',  
          'four',  
          'one',  
          'two',  
          'two',  
          'three',  
          'five',  
          'five',  
          'six',  
          'seven',  
          'six',  
          'seven',  
          'one',  
          'one',  
          'one',  
          'ten',  
          'eight',  
          'ten',  
          'nine',  
          'eleven',  
          'ten',  
          'ten',  
          'nine']
```

```
In [63]: mylist1 = set(mylist) # Unique values in a list
mylist1 = list(mylist1)
mylist1
```

```
Out[63]: ['nine',
          'one',
          'eight',
          'two',
          'seven',
          'ten',
          'four',
          'five',
          'three',
          'eleven',
          'six']
```

```
In [60]: # Calculate frequenc of each word
count1 = [0] * len(mylist1)
mydict5 = dict()
for i in range(len(mylist1)):
    for j in range(len(mylist)):
        if mylist1[i] == mylist[j]:
            count1[i] += 1
    mydict5[mylist1[i]] = count1[i]
print(mydict5)
```

```
{'nine': 2, 'one': 5, 'eight': 1, 'two': 3, 'seven': 2, 'ten': 4, 'four': 1, 'five': 2, 'three': 2, 'eleven': 1, 'six': 2}
```

Operators

- Operators are special symbols in Python which are used to perform operations on variables/values.

Arithmetic Operators

```
In [81]: a = 5
b = 2

x = 'Asif'
y = 'Bhat'

# Addition
c = a + b
print('Addition of {} and {} will give :- {}'.format(a,b,c))

#Concatenate string using plus operator
z = x+y
print ('Concatenate string \'x\' and \'y\' using \'+\' operaotr :- {}'.format(z))

# Subtraction
c = a - b
print('Subtracting {} from {} will give :- {}'.format(b,a,c))

# Multiplication
c = a * b
print('Multiplying {} and {} will give :- {}'.format(a,b,c))

# Division
c = a / b
print('Dividing {} by {} will give :- {}'.format(a,b,c))

# Modulo of both number
c = a % b
print('Modulo of {} , {} will give :- {}'.format(a,b,c))

# Power
c = a ** b
print('{} raised to the power {} will give :- {}'.format(a,b,c))

# Division(floor)
c = a // b
print('Floor division of {} by {} will give :- {}'.format(a,b,c))
```

Addition of 5 and 2 will give :- 7

Concatenate string 'x' and 'y' using '+' operaoatr :- AsifBhat

Subtracting 2 from 5 will give :- 3

Multiplying 5 and 2 will give :- 10

Dividing 5 by 2 will give :- 2.5

Modulo of 5 , 2 will give :- 1

5 raised to the power 2 will give :- 25

Floor division of 5 by 2 will give :- 2

Comparison Operators

Comparison operators are used to compare values.

```
In [84]: x = 20
y = 30

print('Is x greater than y :- ',x>y)

print('\nIs x less than y :- ',x<y)

print('\nIs x equal to y :- ',x==y)

print('\nIs x not equal to y :- ',x!=y)

print('\nIs x greater than or equal to y :- ',x>=y)

print('\nIs x less than or equal to y :- ',x<=y)
```

Is x greater than y :- False

Is x less than y :- True

Is x equal to y :- False

Is x not equal to y :- True

Is x greater than or equal to y :- False

Is x less than or equal to y :- True

```
In [87]: a = 'Asif'
b = 'Bhat'
c = 'Asif'

a == b , a ==c , a != b # Comparison operators on string
```

Out[87]: (False, True, True)

Logical Operators

```
In [92]: x = True
y = False

print('Logical AND operation :- ',x and y) # True if both values are true
print('Logical OR operation :- ',x or y) # True if either of the values is true
print('NOT operation :- ',not x) # True if operand is false
```

```
Logical AND operation :- False
Logical OR operation :- True
NOT operation :- False
```

Bitwise operators

Bitwise operators act on bits and performs bit by bit operation.

```
In [98]: x = 18 # binary form 10010
y = 6 # binary form 00110

print('Bitwise AND operation - {}'.format(x&y))
print('Bitwise OR operation - {}'.format(x|y))
print('Bitwise XOR operation - {}'.format(x^y))
print('Bitwise NOT operation - {}'.format(~x))
print('Bitwise right shift operation - {}'.format(x>>2))
print('Bitwise left shift operation - {}'.format(x<<2))
```

```
Bitwise AND operation - 2
Bitwise OR operation - 22
Bitwise XOR operation - 20
Bitwise NOT operation - -19
Bitwise right shift operation - 4
Bitwise left shift operation - 72
```

Assignment Operators


```
In [120]: x = 10

print('Initialize x with value 10 (x=10)) :- ',x)

x+=20 # x = x+20
print ('Add 20 to x :- ',x)

x-=20 # x = x-20
print ('subtract 20 from x :- ',x)

x/=10 # x = x/10
print ('Divide x by 10 :- ',x)

x*=10 # x = x/10
print ('Multiply x by 10 :- ',x)

x = int(x)

x**=2 # x = x/10
print ('x raised to the power 2 :- ',x)

x%=2
print ('Modulo Division :- ',x)

x = 20

x//=3
print ('Floor Division :- ',x)

x&=2
```

```
print('Bitwise AND :- ',x)

x|=2
print('Bitwise OR :- ',x)

x^=2
print('Bitwise XOR :- ',x)

x = 10

x<<=2
print('Bitwise left shift operation',x)

x>>=2
print('Bitwise right shift operation',x)
```

```
Initialize x with value 10 (x=10)) :- 10
Add 20 to x :- 30
subtract 20 from x :- 10
Divide x by 10 :- 1.0
Multiply x by 10 :- 10.0
x raised to the power 2 :- 100
Modulo Division :- 0
Floor Division :- 6
Bitwise AND :- 2
Bitwise OR :- 2
Bitwise XOR :- 0
Bitwise left shift operation 40
Bitwise right shift operation 10
```

Membership Operators

Membership Operators are used to test whether a value / variable is present in a sequence.

```
In [122]: mystr = 'Asif Ali Bhat'

'Asif' in mystr , 'John' in mystr
```

```
Out[122]: (True, False)
```

```
In [123]: mystr = 'Asif Ali Bhat'

'Asif' not in mystr , 'John' not in mystr
```

```
Out[123]: (False, True)
```

Functions

- A function is a block of organized code written to carry out a specified task.
- Functions help break our program into smaller and modular chunks for better readability.
- Information can be passed into a function as arguments.
- Parameters are specified after the function name inside the parentheses.
- We can add as many parameters as we want. Parameters must be separated with a comma.
- A function may or may not return data.
- In Python a function is defined using the **def** keyword

Parameter VS Argument

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

Three types of functions in Python:-

- **Built-in function** :- Python predefined functions that are readily available for use like `min()` , `max()` , `sum()` , `print()` etc.
- **User-Defined Functions**:- Function that we define ourselves to perform a specific task.
- **Anonymous functions** : Function that is defined without a name. Anonymous functions are also called as lambda functions. They are not declared with the **def** keyword.

Syntax

```
def FunctionName( parameters ):
    """ Function DocString """
    statement(s)
    return [expression]
```

Modularity

```
def CollectData():  
    """Function to collect data"""  
    statement(s)  
  
def CleanData():  
    """Function to clean data"""  
    statement(s)  
  
def ProcessData():  
    """Function to process data"""  
    statement(s)  
  
def ExploreData():  
    """Function to Explore data"""  
    statement(s)  
  
def VisualizaData():  
    """Function to visualize data"""  
    statement(s)
```

```
# Main Program
CollectData()
CleanData()
ProcessData()
ExploreData()
VisualizaData()
```

```
In [582]: def myfunc():
          print("Hello Python Lovers")

          myfunc()
```

Hello Python Lovers

```
In [585]: def details(name,userid,country): # Function to print User details
          print('Name :- ', name)
          print('User ID is :- ', userid)
          print('Country :- ',country)

          details('Asif' , 'asif123' , 'India')
```

Name :- Asif
User ID is :- asif123
Country :- India

```
In [586]: def square (n): #function to find square of a number
          n= n*n
          return n

          square (10)
```

Out[586]: 100

```
In [39]: def even_odd (num): #Even odd test
        """ This function will check whether a number is even or odd"""
        if num % 2 ==0:
            print (num, ' is even number')
        else:
            print (num, ' is odd number')

even_odd(3)
even_odd(4)
print(even_odd.__doc__) # Print function documentation string
```

```
3 is odd number
4 is even number
This function will check whether a number is even or odd
```

```
In [590]: def fullname (firstname , middlename ,lastname): #Concatenate Strings
        fullname = "{} {} {}".format(firstname,middlename,lastname)
        print (fullname)
```

```
fullname('Asif' , 'Ali' , 'Bhat')
```

```
Asif Ali Bhat
```

```
In [591]: def fullname (firstname , middlename ,lastname): #Concatenate Strings
        fullname = "{} {} {}".format(firstname,middlename,lastname)
        print (fullname)
```

```
fullname(lastname = 'Bhat' , middlename='Ali' , firstname='Asif') # Keyword Arguments. Order of the arguments does not matter
```

```
Asif Ali Bhat
```

```
In [592]: fullname ('Asif') # This will throw error as function is expecting 3 arguments.
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-592-d194f8b98253> in <module>
----> 1 fullname ('Asif')
```

```
TypeError: fullname() missing 2 required positional arguments: 'middlename' and 'lastname'
```

```
In [596]: def myfunc(city = 'Mumbai'):
          print('Most Populous City :- ', city)

          myfunc() # When a function is called without an argument it will use default value

          Most Populous City :- Mumbai
```

```
In [26]: var1 = 100 # Variable with Global scope.

          def myfunc():
              print(var1) # Value 100 will be displayed due to global scope of var1

          myfunc()
          print(var1)

          100
          100
```



```
In [27]: def myfunc1():
          var2 = 10 # Variable with Local scope
          print(var2)

def myfunc2():
    print(var2) # This will throw error because var2 has a local scope. Var2 is only accessible in myfunc1()

myfunc1()
myfunc2()
```

10

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-27-6a1c34e80ca2> in <module>
      8
      9 myfunc1()
----> 10 myfunc2()

<ipython-input-27-6a1c34e80ca2> in myfunc2()
      5
      6 def myfunc2():
----> 7     print(var2) # Value 100 will be displayed due to global scope of var1
      8
      9 myfunc1()

NameError: name 'var2' is not defined
```

In [29]: `var1 = 100` # Variable with Global scope.

```
def myfunc():  
    var1 = 99 # Local scope  
    print(var1)
```

```
myfunc()  
print(var1) # The original value of var1 (100) will be retained due to global scope.
```

99
100

In [33]: `list1 = [11,22,33,44,55]`

```
def myfunc(list1):  
    del list1[0]
```

```
print("List1" before calling the function:- ',list1)  
myfunc(list1) # Pass by reference (Any change in the parameter within the function is reflected back in the calling function)  
print("List1" after calling the function:- ',list1)
```

"List1" before calling the function:- [11, 22, 33, 44, 55]
"List1" after calling the function:- [22, 33, 44, 55]

In [34]: `list1 = [11,22,33,44,55]`

```
def myfunc(list1):  
    list1.append(100)
```

```
print("List1" before calling the function:- ',list1)  
myfunc(list1) # Pass by reference (Any change in the parameter within the function is reflected back in the calling function)  
print("List1" after calling the function:- ',list1)
```

"List1" before calling the function:- [11, 22, 33, 44, 55]
"List1" after calling the function:- [11, 22, 33, 44, 55, 100]

```
In [46]: list1 = [11,22,33,44,55]

def myfunc(list1):
    list1 = [10,100,1000,10000] # Link of 'list1' with previous object is broken now as new object is assigned to 'list1'

print("List1" before calling the function:- ',list1)
myfunc(list1) # Pass by reference (Any change in the parameter within the function is reflected back in the calling function)
print("List1" after calling the function:- ',list1)

"List1" before calling the function:- [11, 22, 33, 44, 55]
"List1" after calling the function:- [11, 22, 33, 44, 55]
```

```
In [45]: def swap(a,b):
        temp = a
        a = b      # Link of 'a' with previous object is broken now as new object is assigned to 'a'.
        b = temp    # Link of 'b' with previous object is broken now as new object is assigned to 'b'.

a = 10
b = 20
swap(a,b)
a,b
```

Out[45]: (10, 20)

```
In [601]: def factorial(num): # Calculate factorial of a number using recursive function call.
        if num <=1 :
            return 1
        else:
            return num * factorial(num-1)

factorial(4)
```

Out[601]: 24

```
In [618]: def add(num): # Sum of first n natural numbers
            if num == 0:
                return 0
            else:
                return num + add(num-1)

add(5) # Sum of first five natural numbers (1,2,3,4,5)
```

Out[618]: 15

```
In [12]: def fiboacci(num):
            if num <= 1:
                return num
            if num == 2:
                return 1
            else:
                return(fiboacci(num-1) + fiboacci(num-2))

nums = int(input("How many fibonacci numbers you want to generate -"))

for i in range(nums):
    print(fiboacci(i)) # Generate Fibonacci series
```

```
How many fibonacci numbers you want to generate -10
0
1
1
2
3
5
8
13
21
34
```

args & kwargs

***args**

- When we are not sure about the number of arguments being passed to a function then we can use ***args** as function parameter.
- ***args** allow us to pass the variable number of **Non Keyword Arguments** to function.
- We can simply use an asterisk ***** before the parameter name to pass variable length arguments.
- The arguments are always passed as a tuple.
- We can rename it to anything as long as it is preceded by a single asterisk (*****). It's best practice to keep naming it **args** to make it immediately recognizable.

****kwargs**

- ****kwargs** allows us to pass the variable number of **Keyword Arguments** to the function.
- We can simply use an double asterisk ****** before the parameter name to pass variable length arguments.
- The arguments are passed as a dictionary.
- We can rename it to anything as long as it is preceded by a double asterisk (******). It's best practice to keep naming it **kwargs** to make it immediately recognizable.

def myfunc(positional arguments , ***args** , named arguments , ****kwargs**)

positional
arguments must
come before ***args** ,
named arguments
& ****kwargs**

***args** must come
after positional
arguments & before
named arguments
& ****kwargs**

named arguments
must come after the
positional
arguments & ***args**
but before ****kwargs**

****kwargs** will
always be the
last argument
in the function

Example :- **def TicketDetails**(RequestId, CustomerName , ***args** , status=0 , ****kwargs**)

```
In [578]: def add(a,b,c):  
          return a+b+c  
  
          print(add(10,20,30)) # Sum of two numbers
```

60

```
In [577]: print(add(1,2,3,4)) '''This will throw below error as this function will only take two argumengts.  
          If we want to make argument list dynamic then *args wil come in picture'''
```

File "<ipython-input-577-565d47b69332>", line 2

If we want to make argument list dynamic then *args wil come in picture'''

^

SyntaxError: invalid syntax

```
In [566]: def some_args(arg_1, arg_2, arg_3):  
          print("arg_1:", arg_1)  
          print("arg_2:", arg_2)  
          print("arg_3:", arg_3)  
  
          my_list = [2, 3]  
          some_args(1, *my_list)
```

arg_1: 1

arg_2: 2

arg_3: 3

```
In [524]: def add1(*args):  
          return sum(args)  
  
          print(add(1,2,3))  
          print(add(1,2,3,4)) # *args will take dynamic argument list. So add() function will perform addition of any number of arguments.  
          print(add(1,2,3,4,5))  
          print(add(1,2,3,4,5,6))  
          print(add(1,2,3,4,5,6,7))
```

```
6  
10  
15  
21  
28
```

```
In [561]: list1 = [1,2,3,4,5,6,7]  
          tuple1 = (1,2,3,4,5,6,7)  
  
          add1(*list1) , add1(*tuple1) #tuple & list items will be passed as argument list and sum will be returned for both cases
```

```
Out[561]: (28, 28)
```

```
In [562]: list1 = [1,2,3,4,5,6,7]  
          list2 = [1,2,3,4,5,6,7]  
          list3 = [1,2,3,4,5,6,7]  
          list4 = [1,2,3,4,5,6,7]  
  
          add1(*list1 , *list2 , *list3 , *list4 ) #All four lists are unpacked and each individual item is passed to add1() function
```

```
Out[562]: 112
```

```
In [511]: def UserDetails(*args):  
          print(args)  
  
          UserDetails('Asif' , 7412 , 41102 , 33 , 'India' , 'Hindi')  
  
          ''' For the above example we have no idea about the parameters passed e.g 7412 , 41102 , 33 etc.  
              In such cases we can take help of Keyworded arguments (**kwargs) '''  
  
          ('Asif', 7412, 41102, 33, 'India', 'Hindi')
```

```
In [517]: def UserDetails(**kwargs):  
          print(kwargs)  
  
UserDetails(Name='Asif' , ID=7412 , Pincode=41102 , Age= 33 , Country= 'India' , Language= 'Hindi')  
  
{'Name': 'Asif', 'ID': 7412, 'Pincode': 41102, 'Age': 33, 'Country': 'India', 'Language': 'Hindi'}
```

```
In [519]: def UserDetails(**kwargs):  
          for key,val in kwargs.items():  
              print("{} :- {}".format(key,val))  
  
UserDetails(Name='Asif' , ID=7412 , Pincode=41102 , Age= 33 , Country= 'India' , Language= 'Hindi')  
  
Name :- Asif  
ID :- 7412  
Pincode :- 41102  
Age :- 33  
Country :- India  
Language :- Hindi
```

```
In [523]: mydict = {'Name': 'Asif', 'ID': 7412, 'Pincode': 41102, 'Age': 33, 'Country': 'India', 'Language': 'Hindi'}  
  
UserDetails(**mydict)  
  
Name :- Asif  
ID :- 7412  
Pincode :- 41102  
Age :- 33  
Country :- India  
Language :- Hindi
```



```
In [553]: def UserDetails(licenseNo, *args , phoneNo=0 , **kwargs): # Using all four arguments types
    print('License No :- ', licenseNo)
    j=''
    for i in args:
        j = j+i
    print('Full Name :-',j)
    print('Phone Number:- ',phoneNo)
    for key,val in kwargs.items():
        print("{} :- {}".format(key,val))

name = ['Asif' , ' ' , 'Ali' , ' ' , 'Bhat']
mydict = {'Name': 'Asif', 'ID': 7412, 'Pincode': 41102, 'Age': 33, 'Country': 'India', 'Language': 'Hindi'}

UserDetails('BHT145' , *name , phoneNo=1234567890,**mydict )
```

```
License No :- BHT145
Full Name :- Asif Ali Bhat
Phone Number:- 1234567890
Name :- Asif
ID :- 7412
Pincode :- 41102
Age :- 33
Country :- India
Language :- Hindi
```

```
In [554]: def UserDetails(licenseNo, *args , phoneNo=0, **kwargs): # Using all four arguments types. CORRECT ORDER
    print('Nothing')
```

```
In [557]: def UserDetails(licenseNo, **kwargs , *args): # This will fail. *args MUST come before **kwargs in the argument list
    print('Nothing')
```

```
File "<ipython-input-557-dcd3c92277bc>", line 1
```

```
    def UserDetails(licenseNo, **kwargs , *args): # This will fail. *args MUST come before **kwargs in the argument list
t
```

^

SyntaxError: invalid syntax

```
In [564]: #The below function will fail. Default argument/positional argument (licenseNo) MUST come before Keyword argument(ID)  
def UserDetails(ID = 1, licenseNo, *args):  
    print('Nothing')
```

```
File "<ipython-input-564-8a3e722c7ed7>", line 2
```

```
def UserDetails(ID = 1, licenseNo, *args):
```

```
^
```

SyntaxError: non-default argument follows default argument

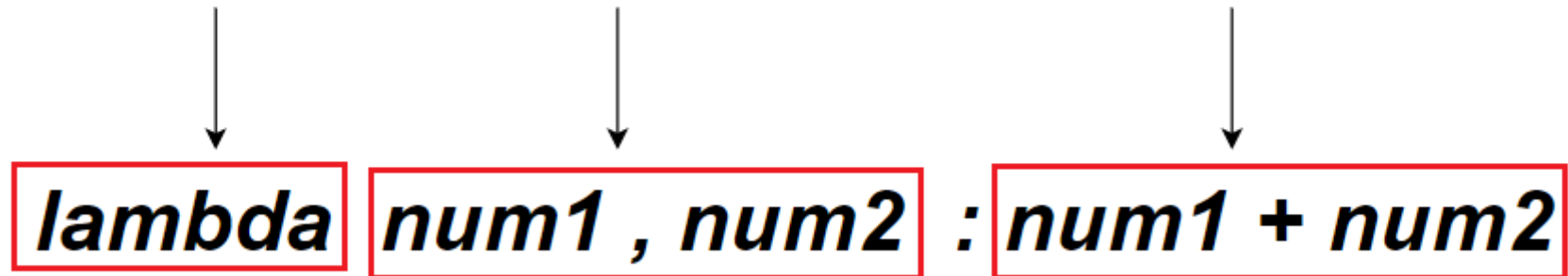
Lambda, Filter, Map and Reduce

Lambda

- A lambda function is an anonymous function (function without a name).
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.
- We use lambda functions when we require a nameless function for a short period of time.

Syntax:-

lambda argument(s) : expression



lambda num1 , num2 : num1 + num2

Filter

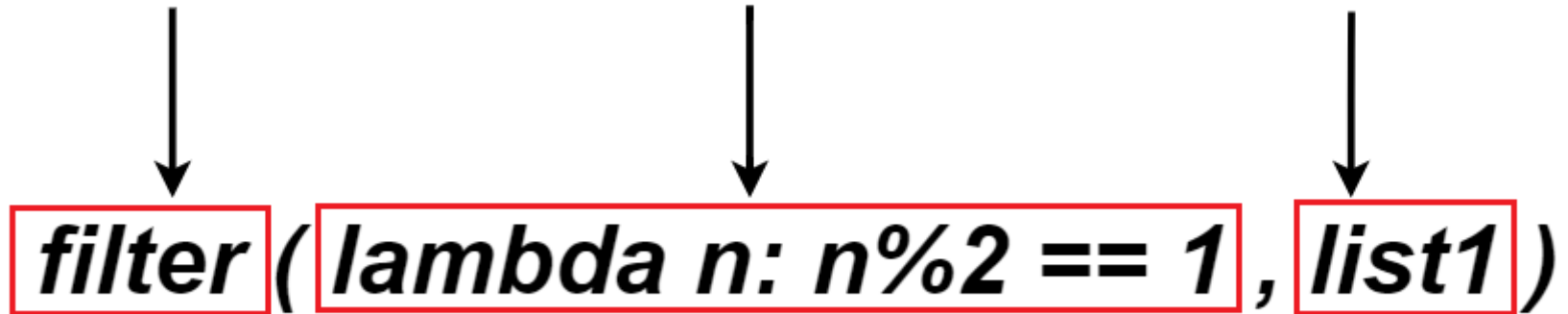
- It is used to filter the iterables/sequence as per the conditions.
- Filter function filters the original iterable and passes the items that returns True for the function provided to filter.
- It is normally used with Lambda functions to filter list, tuple, or sets.

filter() method takes two parameters:

- **function** - function tests if elements of an iterable returns true or false
- **iterable** - Sequence which needs to be filtered, could be sets, lists, tuples, or any iterators

Syntax:

filter (function , iterable)



filter (lambda n: n%2 == 1 , list1)

Map

- The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results.

map() function takes two Parameters :

- **function** : The function to execute for each item of given iterable.
- **iterable** : It is a iterable which is to be mapped.

Returns : Returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Syntax:

map (function , iterable)

*map (lambda num : num*2 , list1)*

Reduce

- The reduce() function is defined in the **functools** python module. The reduce() function receives two arguments, a function and an iterable. However, it doesn't return another iterable, instead it returns a single value.

Working:

- 1) Apply a function to the first two items in an iterable and generate a partial result.
- 2) The function is then called again with the result obtained in **step 1** and the next value in the sequence. This process keeps on repeating until there are items in the sequence.
- 3) The final returned result is returned and printed on console.

Syntax:

reduce (function , iterable)

reduce (lambda a,b : a+b , list1)

```
In [392]: addition = lambda a : a + 10 # This Lambda function adds value 10 to an argument.  
print(addition(5))
```

15

```
In [393]: product = lambda a, b : a * b #This Lambda function takes two arguments (a,b) and returns their product (a*b).  
print(product(5, 6))
```

30

```
In [394]: addition = lambda a, b, c : a + b + c #This Lambda function takes three arguments (a,b,c) and returns their sum (a+b+c)  
print(addition(5, 6, 2))
```

13

```
In [364]: res = (lambda *args: sum(args)) # This Lambda function can take any number of arguments and return thier sum.  
res(10,20) , res(10,20,30,40) , res(10,20,30,40,50,60,70)
```

```
Out[364]: (30, 100, 280)
```

```
In [370]: res1 = (lambda **kwargs: sum(kwargs.values())) # This Lambda function can take any number of arguments and return their sum
res1(a = 10 , b= 20 , c = 30) , res1(a = 10 , b= 20 , c = 30, d = 40 , e = 50)
```

Out[370]: (60, 150)

```
In [386]: res1 = (lambda **kwargs: sum(kwargs.values())) # This Lambda function can take any number of arguments and return their sum
res1(a = 10 , b= 20 , c = 30) , res1(a = 10 , b= 20 , c = 30, d = 40 , e = 50)
```

Out[386]: (60, 150)

```
In [446]: # User defined function to find product of numbers
def product(nums):
    total = 1
    for i in nums:
        total *= i
    return total

# This Lambda function can take any number of arguments and return their product.
res1 = (lambda **kwargs: product(kwargs.values()))
res1(a = 10 , b= 20 , c = 30) , res1(a = 10 , b= 20 , c = 30, d = 40 , e = 50)
```

Out[446]: (6000, 120000000)

```
In [447]: def myfunc(n):
    return lambda a : a + n

add10 = myfunc(10)
add20 = myfunc(20)
add30 = myfunc(30)

print(add10(5))
print(add20(5))
print(add30(5))
```

15
25
35

```
In [437]: list1 = [1,2,3,4,5,6,7,8,9]

def odd(n):
    if n%2 ==1: return True
    else: return False

odd_num = list(filter(odd,list1)) # This Filter function filters list1 and passes all odd numbers to filter().
odd_num
```

Out[437]: [1, 3, 5, 7, 9]

```
In [438]: list1 = [1,2,3,4,5,6,7,8,9]
# The below Filter function filters "list1" and passes all odd numbers using lambda function to filter().
odd_num = list(filter(lambda n: n%2 ==1 ,list1))
odd_num
```

Out[438]: [1, 3, 5, 7, 9]

```
In [439]: def twice(n):
    return n*2

doubles = list(map(twice,odd_num)) # The map function will apply user defined "twice()" function on all items of the list
doubles
```

Out[439]: [2, 6, 10, 14, 18]

```
In [440]: doubles = list(map(lambda n:n*2,odd_num)) # This map function will double all items of the list using lambda function.
doubles
```

Out[440]: [2, 6, 10, 14, 18]


```
In [441]: from functools import reduce

def add(a,b):
    return a+b

sum_all = reduce(add,doubles) # This reduce function will perform sum of all items in the list using user defined "add()"
sum_all
```

Out[441]: 50

```
In [442]: #The below reduce() function will perform sum of all items in the list using lambda function.
sum_all = reduce(lambda a,b : a+b,doubles)
sum_all
```

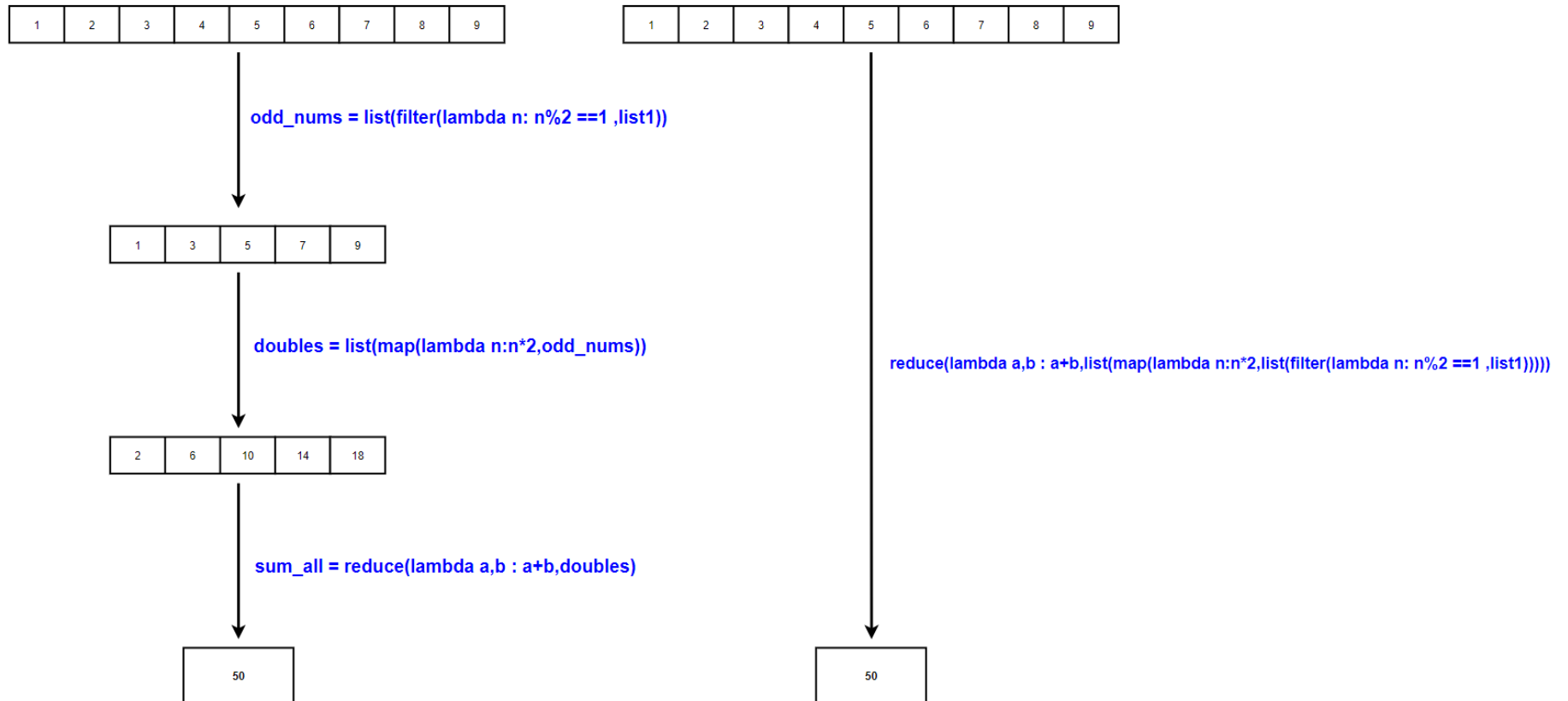
Out[442]: 50

```
In [448]: # Putting all together

sum_all = reduce(lambda a,b : a+b,list(map(lambda n:n*2,list(filter(lambda n: n%2 ==1 ,list1)))))
sum_all
```

Out[448]: 50

Lambda , Filter , Map & Reduce



In []: *# More examples on Map , Filter , Reduce*

```

In [497]: list1 = [1,2,3,4,5,6,7,8,9,10]
even = list(filter(lambda n: n%2 ==0 ,list1)) # Filter even numbers from the list
odd = list(filter(lambda n: n%2 !=0 ,list1)) # Filter odd numbers from the list

print('-----')
print(even)
print(odd)
print('-----')

list2 = ['one' , 'TWO' , 'three' , 'FOUR']

upper = list(filter(lambda x: x.isupper() , list2)) # filter uppercase strings from the list
lower = list(filter(lambda x: x.islower() , list2)) # filter lowercase strings from the list

print(upper)
print(lower)
print('-----')

list3 = ['one' , 'two2' , 'three3' , '88' , '99' , '102']

numeric = list(filter(lambda x:x.isnumeric(), list3)) # filter numbers from the list

alpha = list(filter(lambda x:x.isalpha(), list3)) # filter character strings from the list

alphanum = list(filter(lambda x:x.isalnum(), list3)) # filter numbers & character strings from the list

print(alpha)
print(numeric)
print(alphanum)
print('-----')

#Vowel Test

```

```

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

```

```
-----  
['TWO', 'FOUR']  
['one', 'three']  
-----  
['one']  
['88', '99', '102']  
['one', 'two2', 'three3', '88', '99', '102']  
-----
```

```
In [501]: list1 = [1,2,3,4]
list2 = [5,6,7,8]

def double(x):
    return x+x

def add(x,y):
    return x+y

def square(x):
    return x*x

print('-----')

print(list(map(double, list1))) # Double each number using map & User defined function
print(list(map(add, list1, list2))) # add two items using map & User defined function
print(list(map(square, list1))) #Square numbers using map & User defined function

print('-----')

print(list(map(lambda x: x + x, list1))) # Double each number using map & Lambda
print(list(map(lambda x, y: x + y, list1, list2))) # add two items using map & Lambda
print(list(map(lambda x: x*x, list1))) #Square numbers using map & Lambda

print('-----')
```

```
-----
[2, 4, 6, 8]
[6, 8, 10, 12]
[1, 4, 9, 16]
-----
[2, 4, 6, 8]
[6, 8, 10, 12]
[1, 4, 9, 16]
-----
```

```
In [459]: list2 = [1,2,3,4]

product = reduce (operator.mul,list2) # Product of all numbers in a list

add = reduce(operator.add,list2) # Add all numbers in the list

concat_str = reduce(operator.add , ['Python' , ' ' , 'Rocks']) # Concatenate string using reduce

prod = reduce(operator.mul,['Hello ' , 3]) #Repeat a string multiple times

min_num = reduce(lambda a, b: a if a < b else b, list2) # Minimum number in the list using reduce () & lambda

max_num = reduce(lambda a, b: a if a > b else b, list2) # Maximum number in the list using reduce () & lambda

print(product)

print(add)

print(concat_str)

print(prod)

print(min_num)

print(max_num)
```

24

10

Python Rocks

Hello Hello Hello

1

4

```
In [461]: def min_func(a, b):
           return a if a < b else b

def max_func(a, b):
    return a if a > b else b

min_num = reduce(min_func, list2) # Minimum number in the List using reduce () & User defined min function

max_num = reduce(max_func, list2) # Maximum number in the List using reduce () & User defined min function

min_num , max_num
```

Out[461]: (1, 4)

```
In [474]: print('-----')
print(reduce(lambda a, b: bool(a and b), [0, 0, 1, 0, 0])) # Returns True if all values in the list are True
print(reduce(lambda a, b: bool(a and b), [2, 3, 1, 5, 6])) # Returns True if all items in the list are True
print(reduce(lambda a, b: bool(a and b), [8, 9, 1, 0, 9])) # Returns True if all values in the list are True

print('-----')

print(reduce(lambda a, b: bool(a or b), [0, 0, 0, 0, 0])) # Returns True if any item in the list is True
print(reduce(lambda a, b: bool(a or b), [2, 3, 1, 5, 6])) # Returns True if any item in the list is True
print(reduce(lambda a, b: bool(a or b), [8, 9, 1, 0, 9])) # Returns True if any item in the list is True

print('-----')
```

```
-----
False
True
False
-----
False
True
True
-----
```

Classes & Objects

- A Class is an object constructor or a "blueprint" for creating objects.
- Objects are nothing but an encapsulation of variables and functions into a single entity.
- Objects get their variables and functions from classes.
- To create a class we use the keyword **class**.
- The first string inside the class is called docstring which gives the brief description about the class.
- All classes have a function called `__init__()` which is always executed when the class is being initiated.
- We can use `__init__()` function to assign values to object properties or other operations that are necessary to perform when the object is being created
- The **self** parameter is a reference to the current instance of the class and is used to access class variables.
- **self** must be the first parameter of any function in the class
- The **super()** builtin function returns a temporary object of the superclass that allows us to access methods of the base class.
- **super()** allows us to avoid using the base class name explicitly and to enable multiple inheritance.

Syntax


```
class myclass:
    "DocString"
    def __init__(self, var1, var2)
        self.var1 = var1
        self.var2 = var2
        .
        .
        .
    def myfunc1(self):
        print(self.var1)
        print(self.var2)

    def myfunc2(self)
        .
        .
        .
```

```
In [49]: # Create a class with property "var1"
class myclass:
    var1 = 10

obj1 = myclass() # Create an object of class "myclass()"
print(obj1.var1)

10
```

```
In [70]: # Create an employee class
class Employee:
    def __init__(self, name, empid): # __init__() function is used to assign values for name and empid
        self.name = name
        self.empid = empid
    def greet(self): # Class Method
        print("Thanks for joining ABC Company {}".format(self.name))

emp1 = Employee("Asif", 34163) # Create an employee object

print('Name :- ', emp1.name)
print('Employee ID :- ', emp1.empid)
emp1.greet()

Name :- Asif
Employee ID :- 34163
Thanks for joining ABC Company Asif!!
```

```
In [71]: emp1.name = 'Basit' # Modify Object Properties
emp1.name
```

```
Out[71]: 'Basit'
```

```
In [72]: del emp1.empid    # Delete Object Properties
emp1.empid
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-72-b111c8b828fc> in <module>
      1 del emp1.empid    # Delete Object Properties
----> 2 emp1.empid

AttributeError: 'Employee' object has no attribute 'empid'
```

```
In [73]: del emp1 # Delete the object
emp1
```

```
-----
NameError                                    Traceback (most recent call last)
<ipython-input-73-db2cb77ec9fb> in <module>
      1 del emp1 # Delete the object
----> 2 emp1

NameError: name 'emp1' is not defined
```

```
In [75]: emp2 = Employee("Michael", 34162) # Create an employee object

print('Name :- ',emp2.name)
print('Employee ID :- ',emp2.empid)
emp2.greet()
```

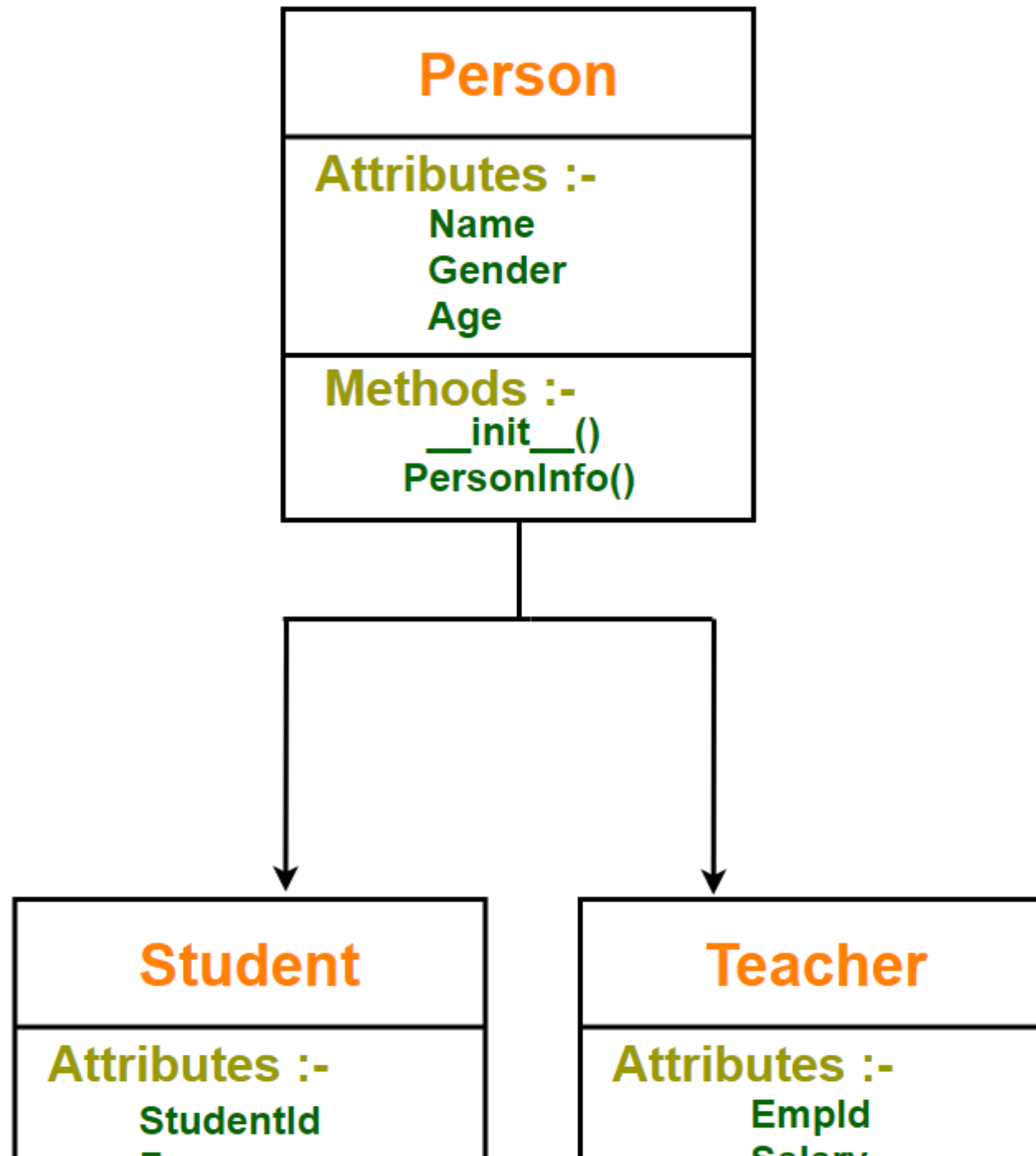
```
Name :- Michael
Employee ID :- 34162
Thanks for joining ABC Company Michael!!
```

```
In [77]: emp2.country = 'India' #instance variable can be created manually
emp2.country
```

```
Out[77]: 'India'
```

Inheritance

- Inheritance is a powerful feature in object oriented programming.
- Inheritance provides code reusability in the program because we can use an existing class (Super Class/ Parent Class / Base Class) to create a new class (Sub Class / Child Class / Derived Class) instead of creating it from scratch.
- The child class inherits data definitions and methods from the parent class which facilitates the reuse of features already available. The child class can add few more definitions or redefine a base class method.
- Inheritance comes into picture when a new class possesses the '**IS A**' relationship with an existing class. E.g Student is a person. Hence person is the base class and student is derived class.



Fees	Salary
Methods :- __init__() StudentInfo()	Methods :- __init__() TeacherInfo()

```
In [163]: class person:    # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))

class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        person.__init__(self,name,age,gender)
        self.studentid = studentid
        self.fees = fees

    def StudentInfo(self):
        print('Student ID :- {}'.format(self.studentid))
        print('Fees :- {}'.format(self.fees))

class teacher(person): # Child Class
    def __init__(self,name,age,gender,empid,salary):
        person.__init__(self,name,age,gender)
        self.empid = empid
        self.salary = salary

    def TeacherInfo(self):
        print('Employee ID :- {}'.format(self.empid))
        print('Salary :- {}'.format(self.salary))

stud1 = student('Asif' , 24 , 'Male' , 123 , 1200)
print('Student Details')
print('-----')
```

```
stud1.PersonInfo()    # PersonInfo() method present in Parent Class will be accessible by child class
stud1.StudentInfo()
print()

teacher1 = teacher('Basit' , 36 , 'Male' , 456 , 80000)
print('Employee Details')
print('-----')
teacher1.PersonInfo()    # PersonInfo() method present in Parent Class will be accessible by child class
teacher1.TeacherInfo()
```

Student Details

Name :- Asif

Age :- 24

Gender :- Male

Student ID :- 123

Fees :- 1200

Employee Details

Name :- Basit

Age :- 36

Gender :- Male

Employee ID :- 456

Salary :- 80000

In [182]: *# super() builtin function allows us to access methods of the base class.*

```
class person:    # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))

class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        super().__init__(name,age,gender)
        self.studentid = studentid
        self.fees = fees

    def StudentInfo(self):
        super().PersonInfo()
        print('Student ID :- {}'.format(self.studentid))
        print('Fees :- {}'.format(self.fees))

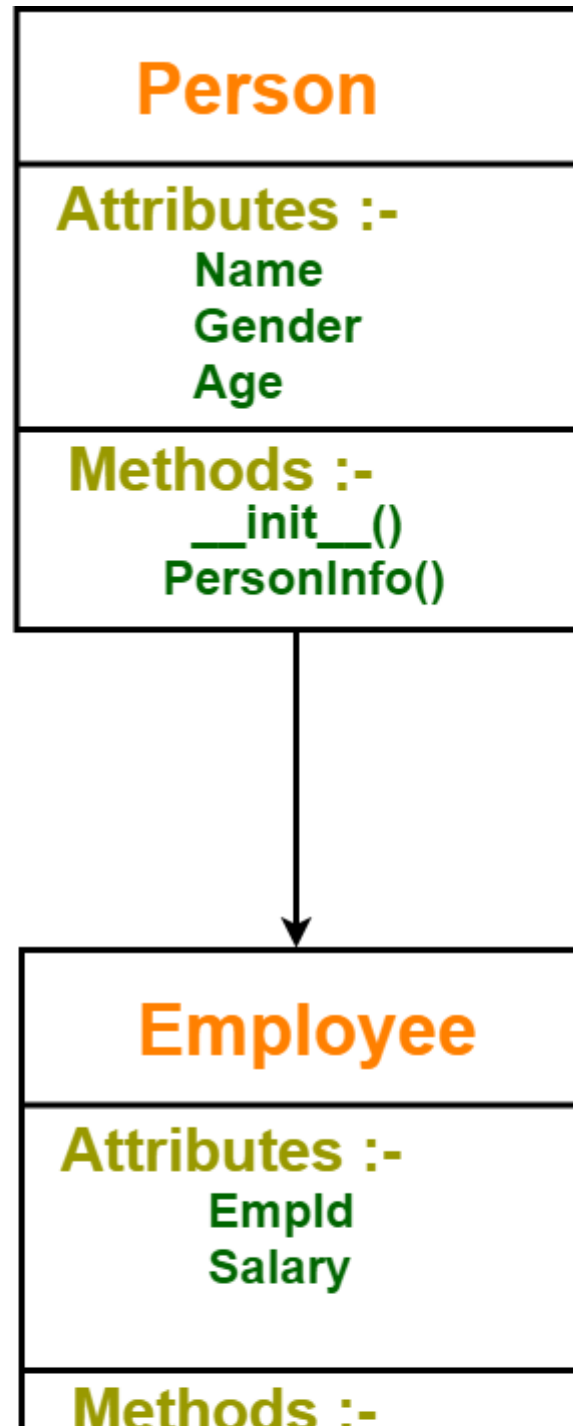
stud = student('Asif' , 24 , 'Male' , 123 , 1200)
stud.StudentInfo()
```

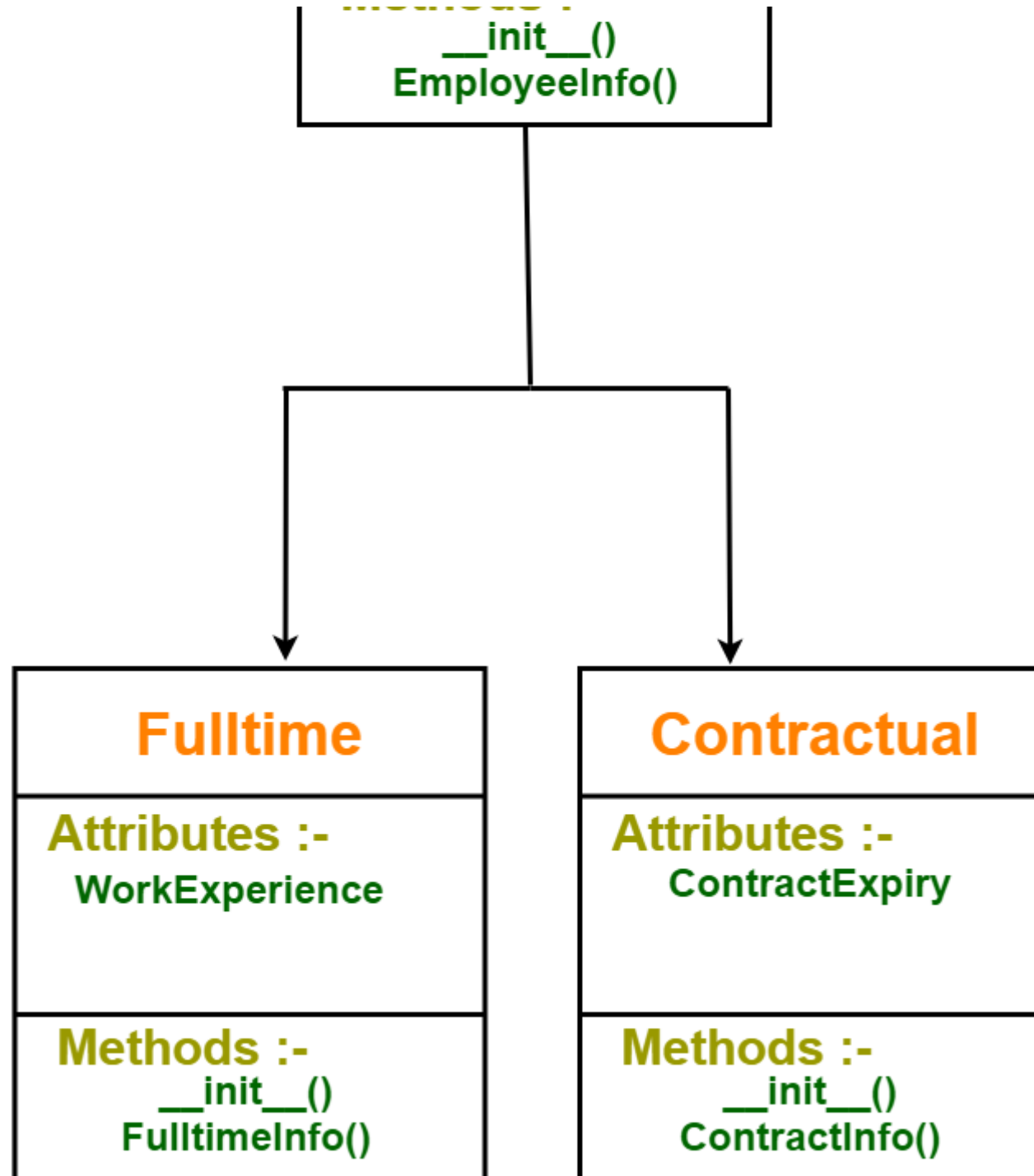
```
Name :- Asif
Age :- 24
Gender :- Male
Student ID :- 123
Fees :- 1200
```

Multi-level Inheritance

- In this type of inheritance, a class can inherit from a child class or derived class.

- Multilevel Inheritance can be of any depth in python






```
In [196]: class person:    # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))

class employee(person): # Child Class
    def __init__(self,name,age,gender,empid,salary):
        person.__init__(self,name,age,gender)
        self.empid = empid
        self.salary = salary

    def employeeInfo(self):
        print('Employee ID :- {}'.format(self.empid))
        print('Salary :- {}'.format(self.salary))

class fulltime(employee): # Grand Child Class
    def __init__(self,name,age,gender,empid,salary,WorkExperience):
        employee.__init__(self,name,age,gender,empid,salary)
        self.WorkExperience = WorkExperience

    def FulltimeInfo(self):
        print('Work Experience :- {}'.format(self.WorkExperience))

class contractual(employee): # Grand Child Class
    def __init__(self,name,age,gender,empid,salary,ContractExpiry):
        employee.__init__(self,name,age,gender,empid,salary)
```

```

        self.ContractExpiry = ContractExpiry

    def ContractInfo(self):
        print('Contract Expiry :- {}'.format(self.ContractExpiry))

print('Contractual Employee Details')
print('*****')
contract1 = contractual('Basit' , 36 , 'Male' , 456 , 80000, '21-12-2021')
contract1.PersonInfo()
contract1.employeeInfo()
contract1.ContractInfo()

print('\n \n')

print('Fulltime Employee Details')
print('*****')
fulltim1= fulltime('Asif' , 22 , 'Male' , 567 , 70000, 12)
fulltim1.PersonInfo()
fulltim1.employeeInfo()
fulltim1.FulltimeInfo()

```

```

Contractual Employee Details
*****
Name :- Basit
Age :- 36
Gender :- Male
Employee ID :- 456
Salary :- 80000
Contract Expiry :- 21-12-2021

```

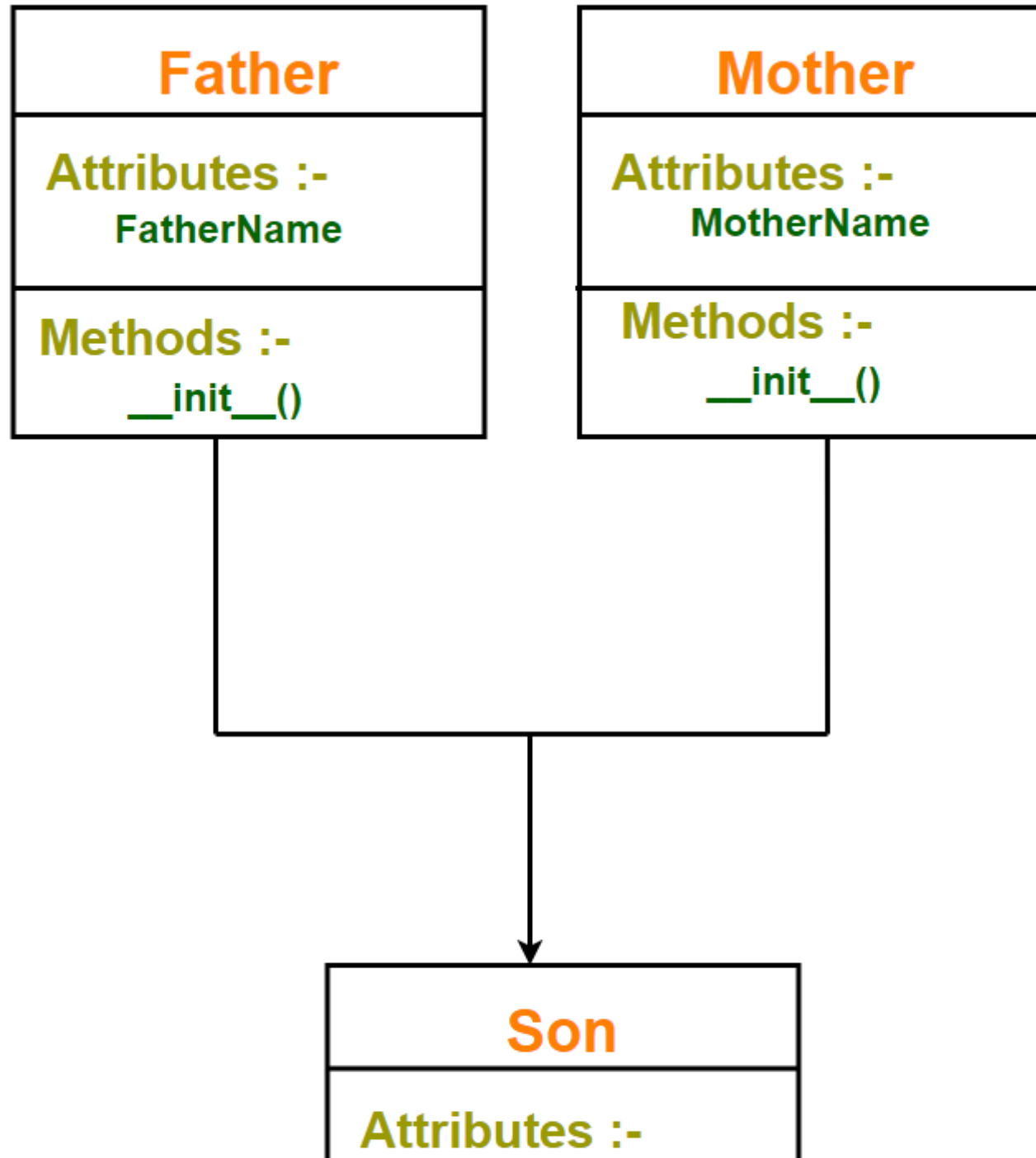
```

Fulltime Employee Details
*****
Name :- Asif
Age :- 22
Gender :- Male
Employee ID :- 567
Salary :- 70000
Work Experience :- 12

```

Multiple Inheritance

- Multiple inheritance is a feature in which a class (derived class) can inherit attributes and methods from more than one parent class.
- The derived class inherits all the features of the base case.



Name
Methods :- Show()

```
In [120]: # Super Class
class Father:
    def __init__(self):
        self.fathername = str()

# Super Class
class Mother:
    def __init__(self):
        self.mothername = str()

# Sub Class
class Son(Father, Mother):
    name = str()
    def show(self):
        print('My Name :- ',self.name)
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

s1 = Son()
s1.name = 'Bill'
s1.fathername = "John"
s1.mothername = "Kristen"
s1.show()
```

```
My Name :- Bill
Father : John
Mother : Kristen
```

```
In [215]: class Date:
          def __init__(self,date):
              self.date = date

          class Time:
              def __init__(self,time):
                  self.time = time

          class timestamp(CurrentDate,CurrentTime):
              def __init__(self,date,time):
                  CurrentDate.__init__(self,date)
                  CurrentTime.__init__(self,time)
                  DateTime = self.date + ' ' + self.time
                  print(DateTime)

          datetime1 = timestamp( '2020-08-09', '23:48:55')
```

2020-08-09 23:48:55

Method Overriding

- Overriding is a very important part of object oriented programming because it makes inheritance exploit its full power.
- Overriding is the ability of a class (Sub Class / Child Class / Derived Class) to change the implementation of a method provided by one of its parent classes.
- When a method in a subclass has the same name, same parameter and same return type as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- The version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

```
In [173]: class person:    # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def greet(self):
        print("Hello Person")

class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        person.__init__(self,name,age,gender)
        self.studentid = studentid
        self.fees = fees

    def greet(self):
        print("Hello Student")

stud = student('Gabriel' , 56 , 'Male' , 45 , 345678)
stud.greet() # greet() method defined in subclass will be triggered as "stud" is an object of child class

person1 = person('Gabriel' , 56 , 'Male')
person1.greet() # greet() method defined in superclass will be triggered because "person1" is an object of parent class
```

Hello Student

Hello Person

Container

- Containers are data structures that hold data values.
- They support membership tests which means we can check whether a value exists in the container or not.
- Generally containers provide a way to access the contained objects and to iterate over them.

- Examples of containers include tuple, list, set, dict, str

```
In [124]: list1 = ['asif' , 'john' , 'Michael' , 'Basit']  
  
         'asif' in list1  # Membership check using 'in' operator
```

Out[124]: True

```
In [128]: assert 'john' in list1  # If the condition returns true the program does nothing and move to the next line of code
```

```
In [127]: assert 'john1' in list1 # If the condition returns false, Assert will stop the program and throws an AssertionError.
```

```
-----  
AssertionError                                Traceback (most recent call last)  
<ipython-input-127-f7bcea8c4682> in <module>  
----> 1 assert 'john1' in list1  
  
AssertionError:
```

```
In [130]: mydict = {'Name': 'Asif' , 'ID': 12345 , 'DOB': 1991 , 'Address' : 'Hilsinki'}  
         mydict
```

Out[130]: {'Name': 'Asif', 'ID': 12345, 'DOB': 1991, 'Address': 'Hilsinki'}

```
In [131]: 'Asif' in mydict # Dictionary membership will always check the keys
```

Out[131]: False

```
In [132]: 'Name' in mydict # Dictionary membership will always check the keys
```

Out[132]: True

```
In [133]: 'DOB' in mydict
```

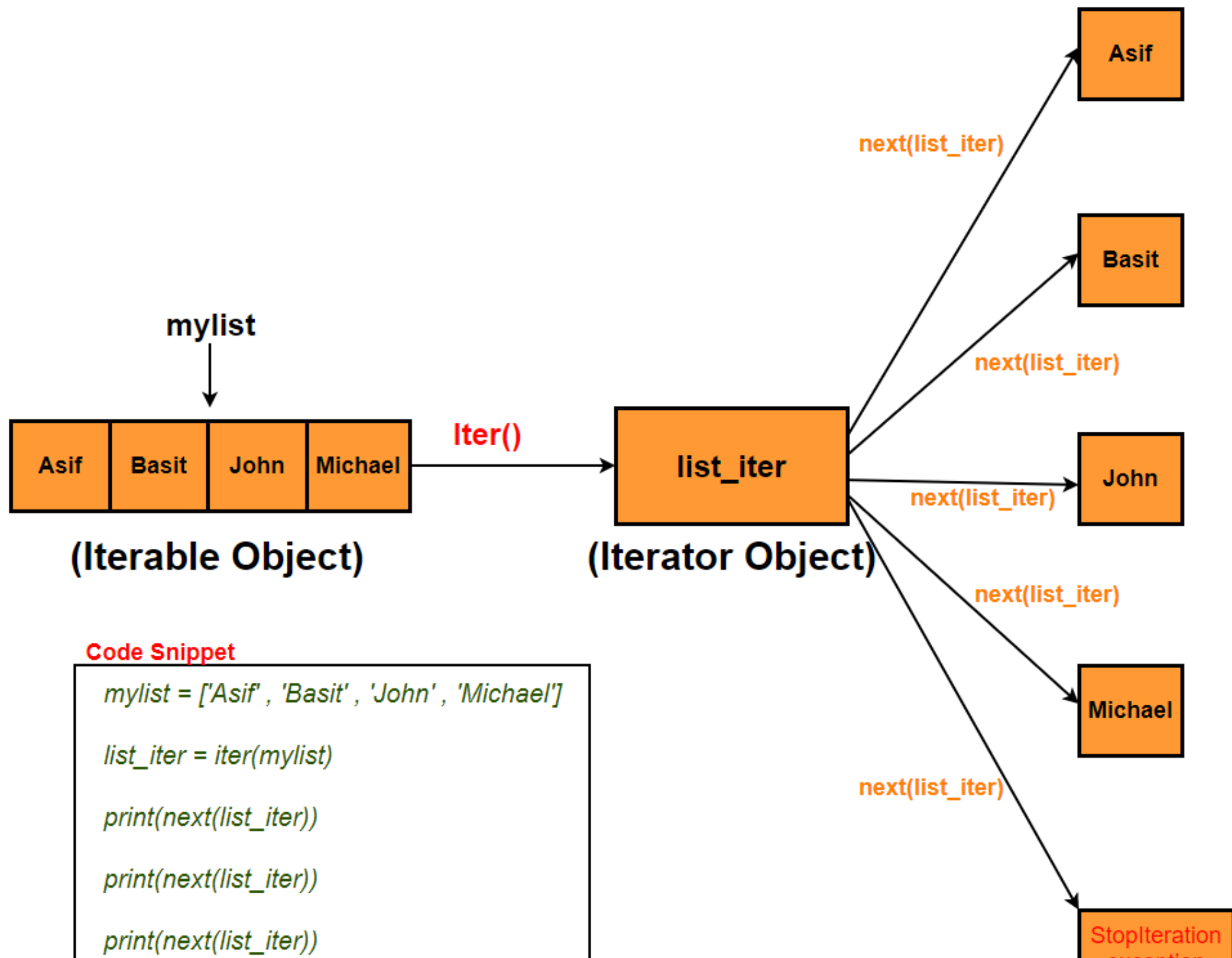
Out[133]: True

```
In [134]: mystr = 'asifbhat'
          'as' in mystr  # Check if substring is present
```

```
Out[134]: True
```

Iterable & Iterator

- An **iterable** is an object that can be iterated upon. It can return an iterator object with the purpose of traversing through all the elements of an iterable.
- An iterable object implements `__iter__()` which is expected to return an iterator object. The iterator object uses the `__next__()` method. Every time `next()` is called next element in the iterator stream is returned. When there are no more elements available **StopIteration exception** is encountered. So any object that has a `__next__()` method is called an **iterator**.
- Python lists, tuples, dictionaries and sets are all examples of iterable objects.



```
print(next(list_iter))
```

```
print(next(list_iter))
```

```
In [236]: mylist = ['Asif' , 'Basit' , 'John' , 'Michael']
list_iter = iter(mylist) # Create an iterator object using iter()
print(next(list_iter))   # return first element in the iterator stream
print(next(list_iter))   # return next element in the iterator stream
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
```

Asif
Basit
John
Michael

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-236-a2095e242a65> in <module>
      5 print(next(list_iter))
      6 print(next(list_iter))
----> 7 print(next(list_iter))
```

StopIteration:

```
In [238]: mylist = ['Asif' , 'Basit' , 'John' , 'Michael']
list_iter = iter(mylist) # Create an iterator object using iter()
print(list_iter.__next__()) # return first element in the iterator stream
print(list_iter.__next__()) # return next element in the iterator stream
print(list_iter.__next__())
print(list_iter.__next__())
```

Asif
Basit
John
Michael


```
In [247]: mylist = ['Asif' , 'Basit' , 'John' , 'Michael']  
list_iter = iter(mylist)      # Create an iterator object using iter()  
for i in list_iter:  
    print(i)
```

Asif
Basit
John
Michael

```
In [241]: # Looping Through an Iterable (list) using for loop
```

```
mylist = ['Asif' , 'Basit' , 'John' , 'Michael']  
  
for i in mylist:  
    print(i)
```

Asif
Basit
John
Michael

```
In [242]: # Looping Through an Iterable (tuple) using for loop
```

```
mytuple = ('Asif' , 'Basit' , 'John' , 'Michael')  
  
for i in mytuple:  
    print(i)
```

Asif
Basit
John
Michael

In [243]: *# Looping Through an Iterable (string) using for loop*

```
mystr = "Hello Python"
```

```
for i in mystr:  
    print(i)
```

H
e
l
l
o

P
y
t
h
o
n

In [255]: *# This iterator produces all natural numbers from 1 to 10.*

```
class myiter:
    def __init__(self):
        self.num = 0

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= 10:
            val = self.num
            self.num += 1
            return val
        else:
            raise StopIteration

mynum = myiter()
iter1 = iter(mynum)
for i in iter1:
    print(i)
```

1
2
3
4
5
6
7
8
9
10

In [256]: *# This iterator will produce odd numbers*

```
class myiter:
    def __init__(self):
        self.num = 0

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= 20 :
            val = self.num
            self.num += 2
            return val
        else:
            raise StopIteration

myodd = myiter()
iter1 = iter(myodd)
for i in iter1:
    print(i)
```

1
3
5
7
9
11
13
15
17
19

In [257]: *# This iterator will produce fibonacci numbers*

```
class myfibonacci:
    def __init__(self):
        self.prev = 0
        self.cur = 0

    def __iter__(self):
        self.prev = 0
        self.cur = 1
        return self

    def __next__(self):
        if self.cur <= 50:
            val = self.cur
            self.cur += self.prev
            self.prev = val
            return val
        else:
            raise StopIteration

myfibo = myfibonacci()
iter1 = iter(myfibo)
for i in iter1:
    print(i)
```

1
1
2
3
5
8
13
21
34

Generator

- Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.
- It is a special type of function which returns an iterator object.
- In a generator function, a **yield** statement is used rather than a **return** statement.
- The generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.
- The difference between **yield** and **return** is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.
- Methods like `__iter()` and `__next()` are implemented automatically in generator function.
- Simple generators can be easily created using **generator expressions**. Generator expressions create anonymous generator functions like lambda.
- The syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces one item at a time which is more memory efficient than list comprehension.

In [258]: *# Simple generator function that will generate numbers from 1 to 5.*

```
def mygen():
    n = 1
    yield n

    n += 1
    yield n

    n += 1
    yield n

    n += 1
    yield n

    n += 1
    yield n

mygen1 = mygen()

print(next(mygen1))
print(next(mygen1))
print(next(mygen1))
print(next(mygen1))
print(next(mygen1)) #Function will terminate here as all 5 values have been returned.
print(next(mygen1)) # As function is already terminated, StopIteration is raised automatically.
```

1
2
3
4
5

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-258-4c1c399db6dd> in <module>
      24 print(next(mygen1))
      25 print(next(mygen1))
----> 26 print(next(mygen1))
```

StopIteration:

In [272]: *# Simple generator function that will generate natural numbers from 1 to 20.*

```
def mygen():  
    for i in range(1,20):  
        yield i
```

```
mygen1 = mygen()
```

```
for i in mygen1:  
    print(i)
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

In [274]: `num = list(mygen())` *# Store all values generated by generator function in a list*
num

Out[274]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

In [275]: *# Simple generator function that will generate even numbers from 1 to 20.*

```
def mygen():  
    for i in range(1,20):  
        if i%2 == 0:  
            yield i
```

```
mygen1 = mygen()
```

```
for i in mygen1:  
    print(i)
```

2
4
6
8
10
12
14
16
18

In [276]: *# This Generator function will generate ten numbers of fibonacci series.*

```
def myfibo():  
    num1 , num2 = 0,1  
    count = 0  
    while count < 10:  
        yield num1  
        num1,num2 = num2,num1+num2  
        count+=1
```

```
fibo = myfibo()
```

```
for i in fibo:  
    print(i)
```

```
0  
1  
1  
2  
3  
5  
8  
13  
21  
34
```

In [277]: `list1 = list(myfibo())` *# Store the fibonacci series in a List*
list1

Out[277]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

In [283]: `list2 = [i**2 for i in range(10)]` *# List comprehension*
list2

Out[283]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [280]: `gen2 = (i**2 for i in range(10))` *# Generator expression*
gen2

Out[280]: <generator object <genexpr> at 0x000001EF4B639848>

```
In [282]: print(next(gen2))
          print(next(gen2))
          print(next(gen2))
          print(next(gen2))
          print(next(gen2))
```

```
1
4
9
16
25
```

```
In [288]: gen2 = (i for i in range(40) if i%2 == 0) # Generator expression to generate even numbers
          gen2

          for i in gen2:
              print(i)
```

```
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
```

Decorator

Decorator is very powerful and useful tool in Python as it allows us to wrap another function in order to extend the behavior of wrapped function without permanently modifying it.

In Decorators functions are taken as the argument into another function and then called inside the wrapper function.

Advantages -

- Logging & debugging
- Access control and authentication

```
In [2]: def subtract(num1 , num2):  
        res = num1 - num2  
        print('Result is :- ', res)  
  
        subtract(4,2)  
        subtract(2,4)
```

```
Result is :- 2  
Result is :- -2
```

```
In [6]: ''' We now want subtract() function to always subtract lower number from higher one without modifying this function.  
        So when we pass (2,4) it should perform 4-2 not 2-4. To achieve this we will create a decorator function'''  
  
def sub_decorator(func):  
    def wrapper(num1,num2):  
        if num1 < num2:  
            num1,num2 = num2,num1  
        return func(num1,num2)  
    return wrapper  
  
sub = sub_decorator(subtract)  
  
sub(2,4)
```

```
Result is :- 2
```

```
In [20]: @sub_decorator    # we can use @ syntax for decorating a function in one step
def subtract(num1 , num2):
    res = num1 - num2
    print('Result is :- ', res)
subtract(2,4)
```

Result is :- 2

```
In [60]: def InstallLinux():
          print('Linux installation has started \n')

def InstallWindows():
    print('Windows installation has started \n')

def InstallMac():
    print('Mac installation has started \n')

InstallLinux()
InstallWindows()
InstallMac()

print()

''' Now suppose if we want to print message :- "Please accept terms & conditions" before every installation
    then easy way will be to create one decorator function which will present this message instead of modifying all func

def InstallDecorator(func):
    def wrapper():
        print('Please accept terms & conditions')
        return func()
    return wrapper()

@InstallDecorator    # we can use @ syntax for decorating a function in one step
def InstallLinux():
    print('Linux installation has started \n')

@InstallDecorator
def InstallWindows():
    print('Windows installation has started \n ')

@InstallDecorator
def InstallMac():
    print('Mac installation has started \n')
```

Linux installation has started

Windows installation has started

Mac installation has started

Please accept terms & conditions
Linux installation has started

Please accept terms & conditions
Windows installation has started

Please accept terms & conditions
Mac installation has started

In [69]: *# Apply multiple decorator on a single function*

```
def InstallDecorator1(func):
    def wrapper():
        print('Please accept terms & conditions...\n')
        func()
    return wrapper

def InstallDecorator2(func):
    def wrapper():
        print('Please enter correct license key...\n')
        return func()
    return wrapper

def InstallDecorator3(func):
    def wrapper():
        print('Please enter partitioning choice...\n')
        return func()
    return wrapper

@InstallDecorator1
@InstallDecorator2
@InstallDecorator3
def InstallLinux():
    print('Linux installation has started \n')

InstallLinux()
```

Please accept terms & conditions...

Please enter correct license key...

Please enter partitioning choice...

Linux installation has started

File Management

Python has several built-in modules and functions for creating, reading, updating and deleting files.

Order of File Operation



Open File

```
In [69]: fileobj = open('test1.txt') # Open file in read/text mode
```

```
In [70]: fileobj = open('test1.txt', 'r') # Open file in read mode
```

```
In [71]: fileobj = open('test1.txt', 'w') # Open file in write mode
```

```
In [72]: fileobj = open('test1.txt', 'a') # Open file in append mode
```

Close File

```
In [73]: fileobj.close()
```

Read File

```
In [84]: fileobj = open('test1.txt')
```

```
In [85]: fileobj.read() #Read whole file
```

```
Out[85]: 'Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.\nIt is a special type of function which returns an iterator object.\nIn a generator function, a yield statement is used rather than a return statement.\nThe generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.\nThe difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.\nMethods like __iter__() and __next__() are implemented automatically in generator function.\nSimple generators can be easily created using generator expressions. Generator expressions create a nonymous generator functions like lambda.\nThe syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces one item at a time which is more memory efficient than list comprehension.'
```

```
In [86]: fileobj.read() #File cursor is already at the end of the file so it won't be able to read anything.
```

```
Out[86]: ''
```

```
In [87]: fileobj.seek(0) # Bring file cursor to initial position.  
fileobj.read()
```

```
Out[87]: 'Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.\nIt is a special type of function which returns an iterator object.\nIn a generator function, a yield statement is used rather than a return statement.\nThe generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.\nThe difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.\nMethods like __iter__() and __next__() are implemented automatically in generator function.\nSimple generators can be easily created using generator expressions. Generator expressions create a nonymous generator functions like lambda.\nThe syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces one item at a time which is more memory efficient than list comprehension.'
```

```
In [88]: fileobj.seek(7) # place file cursor at loc 7
fileobj.read()
```

Out[88]: 'generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.\nIt is a special type of function which returns an iterator object.\nIn a generator function, a yield statement is used rather than a return statement.\nThe generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.\nThe difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.\nMethods like __iter__() and __next__() are implemented automatically in generator function.\nSimple generators can be easily created using generator expressions. Generator expressions create anonymous generator functions like lambda.\nThe syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces one item at a time which is more memory efficient than list comprehension.'

```
In [89]: fileobj.seek(0)

fileobj.read(16) # Return the first 16 characters of the file
```

Out[89]: 'Python generator'

```
In [90]: fileobj.tell() # Get the file cursor position
```

Out[90]: 16

```
In [91]: fileobj.seek(0)

print(fileobj.readline()) # Read first line of a file.

print(fileobj.readline()) # Read second line of a file.

print(fileobj.readline()) # Read third line of a file.
```

Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.

It is a special type of function which returns an iterator object.

In a generator function, a yield statement is used rather than a return statement.

```
In [92]: fileobj.seek(0)

fileobj.readlines() # Read all lines of a file.
```

```
Out[92]: ['Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead
of returning the entire sequence at once.\n',
'It is a special type of function which returns an iterator object.\n',
'In a generator function, a yield statement is used rather than a return statement.\n',
'The generator function cannot include the return keyword. If we include it then it will terminate the execution of the
function.\n',
'The difference between yield and return is that once yield returns a value the function is paused and the control is
transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return
statement value is returned and the execution of the function is terminated.\n',
'Methods like __iter__() and __next__() are implemented automatically in generator function.\n',
'Simple generators can be easily created using generator expressions. Generator expressions create anonymous generator
functions like lambda.\n',
'The syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets
are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces
one item at a time which is more memory efficient than list comprehension.']
```

```
In [93]: # Read first 5 lines of a file using readline()
fileobj.seek(0)

count = 0
for i in range(5):
    if (count < 5):
        print(fileobj.readline())
    else:
        break
    count+=1
```

Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.

It is a special type of function which returns an iterator object.

In a generator function, a yield statement is used rather than a return statement.

The generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.

The difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.

```
In [94]: # Read first 5 lines of a file using readlines()
fileobj.seek(0)

count = 0
for i in fileobj.readlines():
    if (count < 5):
        print(i)
    else:
        break
    count+=1
```

Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.

It is a special type of function which returns an iterator object.

In a generator function, a yield statement is used rather than a return statement.

The generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.

The difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.

Write File

```
In [95]: fileobj = open('test1.txt', 'a')

fileobj.write('THIS IS THE NEW CONTENT APPENDED IN THE FILE') # Append content to the file

fileobj.close()

fileobj = open('test1.txt')

fileobj.read()
```

Out[95]: 'Python generators are easy way of creating iterators. It generates values one at a time from a given sequence instead of returning the entire sequence at once.\nIt is a special type of function which returns an iterator object.\nIn a generator function, a yield statement is used rather than a return statement.\nThe generator function cannot include the return keyword. If we include it then it will terminate the execution of the function.\nThe difference between yield and return is that once yield returns a value the function is paused and the control is transferred to the caller. Local variables and their states are remembered between successive calls. In case of the return statement value is returned and the execution of the function is terminated.\nMethods like __iter__() and __next__() are implemented automatically in generator function.\nSimple generators can be easily created using generator expressions. Generator expressions create a nonymous generator functions like lambda.\nThe syntax for generator expression is similar to that of a list comprehension but the only difference is square brackets are replaced with round parentheses. Also list comprehension produces the entire list while the generator expression produces one item at a time which is more memory efficient than list comprehension.THIS IS THE NEW CONTENT APPENDED IN THE FILE'

```
In [96]: fileobj = open("test1.txt", "w")

fileobj.write("NEW CONTENT ADDED IN THE FILE. PREVIOUS CONTENT HAS BEEN OVERWRITTEN") # overwrite the content in the file

fileobj.close()

fileobj = open('test1.txt')

fileobj.read()
```

Out[96]: 'NEW CONTENT ADDED IN THE FILE. PREVIOUS CONTENT HAS BEEN OVERWRITTEN'

```
In [114]: fileobj = open("test2.txt", "w") # Create a new file

fileobj.write("First Line\n")
fileobj.write("Second Line\n")
fileobj.write("Third Line\n")
fileobj.write("Fourth Line\n")
fileobj.write("Fifth Line\n")
fileobj.close()

fileobj = open('test2.txt')

fileobj.readlines()
```

```
Out[114]: ['First Line\n',
           'Second Line\n',
           'Third Line\n',
           'Fourth Line\n',
           'Fifth Line\n']
```

Delete file

```
In [115]: os.remove("test3.txt") # Delete file
```

```
In [116]: os.remove("test3.txt")
```

```
-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-116-fecc9f240170> in <module>
----> 1 os.remove("test3.txt")
```

```
FileNotFoundError: [WinError 2] The system cannot find the file specified: 'test3.txt'
```

```
In [117]: os.rmdir('folder1/') # Delete folder
```

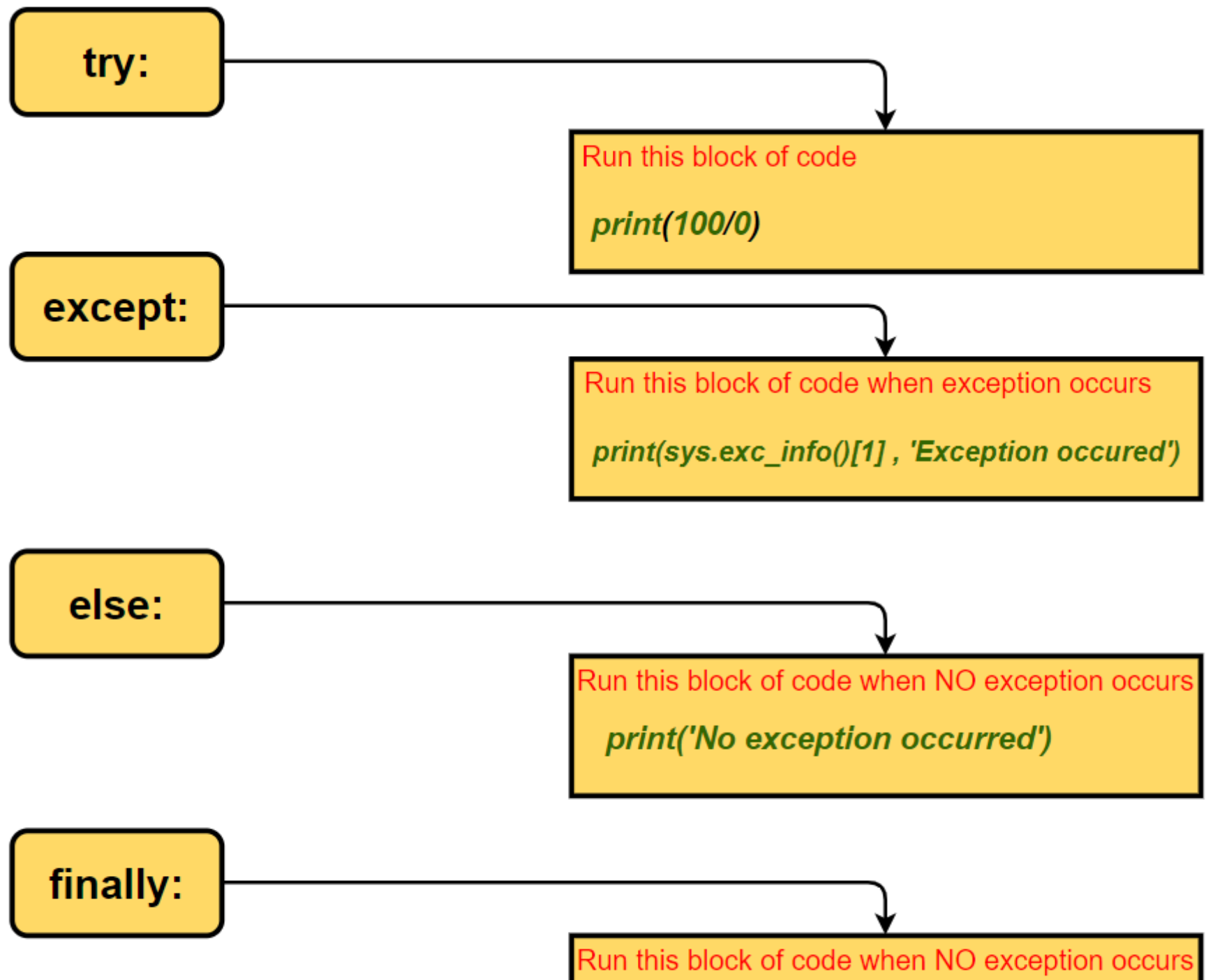


```
In [118]: os.rmdir('folder1/')
```

```
-----  
FileNotFoundError                                Traceback (most recent call last)  
<ipython-input-118-e9e89c9edbf0> in <module>  
----> 1 os.rmdir('folder1/')  
  
FileNotFoundError: [WinError 2] The system cannot find the file specified: 'folder1/'
```

Error & Exception Handling

- Python has many built-in exceptions (ArithmeticError, ZeroDivisionError, EOFError, IndexError, KeyError, SyntaxError, IndentationError, FileNotFoundError etc) that are raised when your program encounters an error.
- When the exception occurs Python interpreter stops the current process and passes it to the calling process until it is handled. If exception is not handled the program will crash.
- Exceptions in python can be handled using a **try** statement. The **try** block lets you test a block of code for errors.
- The block of code which can raise an exception is placed inside the try clause. The code that will handle the exceptions is written in the **except** clause.
- The **finally** code block will execute regardless of the result of the try and except blocks.
- We can also use the **else** keyword to define a block of code to be executed if no exceptions were raised.
- Python also allows us to create our own exceptions that can be raised from the program using the **raise** keyword and caught using the **except** clause. We can define what kind of error to raise, and the text to print to the user.



```
print('Run this block of code always')
```

Code Snippet:-

```
try:  
    print(100/0)  
  
except:  
    print(sys.exc_info()[1] , 'Exception occured')  
  
else:  
    print('No exception occurred')  
  
finally:  
    print('Run this block of code always')
```

Output :-

```
Division by zero Exception occured  
Run this block of code always
```

```
In [130]: try:
            print(100/0) # ZeroDivisionError will be encountered here. So the control will pass to except block

        except:
            print(sys.exc_info()[1] , 'Exception occured') # This statement will be executed

        else:
            print('No exception occurred') # This will be skipped as code block inside try encountered an exception

        finally:
            print('Run this block of code always') # This will be always executed
```

division by zero Exception occured
Run this block of code always

```
In [134]: try:
            print(x) # NameError exception will be encountered as variable x is not defined

        except:
            print('Variable x is not defined')
```

Variable x is not defined

```
In [137]: try:
            os.remove("test3.txt") # FileNotFoundError will be encountered as "test3.txt" is not present in the directory

        except:
            # Below statement will be executed as exception occurred.
            print("BELOW EXCEPTION OCCURED")
            print(sys.exc_info()[1])

        else:
            print('\nNo exception occurred')

        finally:
            print('\nRun this block of code always')
```

BELOW EXCEPTION OCCURED

[WinError 2] The system cannot find the file specified: 'test3.txt'

Run this block of code always

```
In [141]: # Handling specific exceptions
try:
    x = int(input('Enter first number :- '))
    y = int(input('Enter first number :- ')) # If input entered is non-zero the control will move to next line
    print(x/y)
    os.remove("test3.txt")

except NameError:
    print('NameError exception occurred')

except FileNotFoundError:
    print('FileNotFoundError exception occurred')

except ZeroDivisionError:
    print('ZeroDivisionError exception occurred')
```

Enter first number :- 12

Enter first number :- 13

0.9230769230769231

FileNotFoundError exception occurred

In [142]: *# Handling specific exceptions*

```
try:
    x = int(input('Enter first number :- '))
    y = int(input('Enter first number :- ')) # If the input entered is zero the control will move to except block.
    print(x/y)
    os.remove("test3.txt")

except NameError:
    print('NameError exception occurred')

except FileNotFoundError:
    print('FileNotFoundError exception occurred')

except ZeroDivisionError:
    print('ZeroDivisionError exception occurred')
```

```
Enter first number :- 10
Enter first number :- 0
ZeroDivisionError exception occurred
```

In [144]:

```
try:
    x = int(input('Enter first number :- '))
    if x > 50:
        raise ValueError(x) # If value of x is greater than 50 ValueError exception will be encountered.
except:
    print(sys.exc_info()[0])
```

```
Enter first number :- 100
<class 'ValueError'>
```

Built-in Exceptions

In [149]: *# OverflowError - This exception is raised when the result of a numeric calculation is too large*

```
try:
    import math
    print(math.exp(1000))
except OverflowError:
    print (sys.exc_info())
else:
    print ("Success, no error!")
```

(<class 'OverflowError'>, OverflowError('math range error'), <traceback object at 0x000002B2B12EFB88>)

In [150]: *# ZeroDivisionError - This exception is raised when the second operator in a division is zero*

```
try:
    x = int(input('Enter first number :- '))
    y = int(input('Enter first number :- '))
    print(x/y)

except ZeroDivisionError:
    print('ZeroDivisionError exception occurred')
```

Enter first number :- 100
Enter first number :- 0
ZeroDivisionError exception occurred

In [152]: *# NameError - This exception is raised when a variable does not exist*

```
try:
    print(x1)

except NameError:
    print('NameError exception occurred')
```

NameError exception occurred

In [155]: *# AssertionError - This exception is raised when an assert statement fails*

```
try:
    a = 50
    b = "Asif"
    assert a == b
except AssertionError:
    print ("Assertion Exception Raised.")
```

Assertion Exception Raised.

In [157]: *# ModuleNotFoundError - This exception is raised when an imported module does not exist*

```
try:
    import MyModule

except ModuleNotFoundError:
    print ("ModuleNotFoundError Exception Raised.")
```

ModuleNotFoundError Exception Raised.

In [160]: *# KeyError - This exception is raised when key does not exist in a dictionary*

```
try:
    mydict = {1:'Asif', 2:'Basit', 3:'Michael'}
    print (mydict[4])

except KeyError:
    print ("KeyError Exception Raised.")
```

KeyError Exception Raised.

In [162]: *# IndexError - This exception is raised when an index of a sequence does not exist.*

```
try:
    mylist = [1,2,3,4,5,6]
    print (mylist[10])

except IndexError:
    print ("IndexError Exception Raised.")
```

IndexError Exception Raised.

In [165]: *# TypeError - This exception is raised when two different datatypes are combined*

```
try:
    a = 50
    b = "Asif"
    c = a/b
except TypeError:
    print ("TypeError Exception Raised.")
```

TypeError Exception Raised.

In [171]: *# AttributeError: - This exception is raised when attribute reference or assignment fails*

```
try:
    a = 10
    b = a.upper()
    print(b)
except AttributeError:
    print ("AttributeError Exception Raised.")
```

AttributeError Exception Raised.

```
In [ ]: try:
        x = input('Enter first number :- ')

except:
    print('ZeroDivisionError exception occurred')
```

END