**Python**

**Chapter 1 : Getting Started.**

**Features of Python:**

• Python is a high level language.

• It is an interpreted language .

• Python is case-sensitive.

• Python has a rich library of predefined functions.

• Python has application in web development,machine learning,AI/ML any many areas

• Python uses indentation for blocks and nested blocks.

• Python is portable and platform independent, means it can run on various operating systems and hardware platforms.

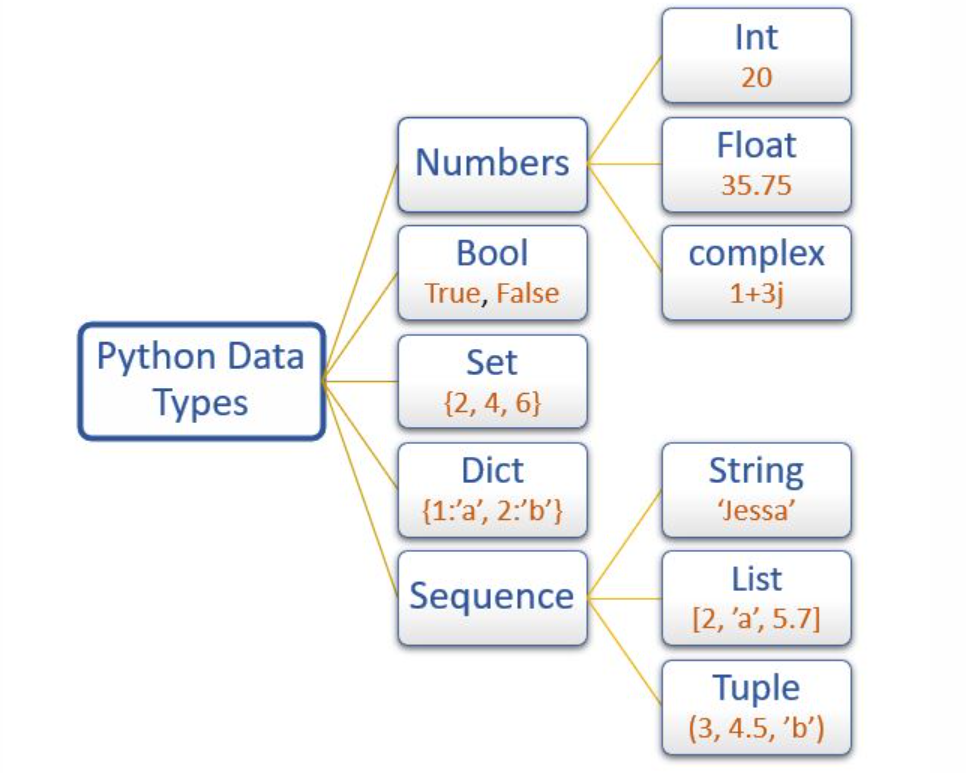
Keywords : Keywords are reserved words.

if , else ,for , while , in , try ,except etc are examples of keywords.

Comments : Comments are used to add a remark or a note in the source code .

In Python, a comment starts with # (hash sign)

**Data types in Python :**



**Sequence :** A Python sequence is an ordered collection of items,where each item is indexed by an integer. The three types of sequence data types available in Python are  
Strings, Lists and Tuples.

**List :**

* List is a sequence of items separated by commas
* The items are enclosed in square brackets [ ]
* The items can be of any type . This means list is heterogenous .
* Example

**list1** = [5, 3.4, "New Delhi", "20C", 45]

**Tuple :**

* Tuple is a sequence of items separated by commas
* The items are enclosed in parenthesis ().
* Once created, we cannot change the tuple, so tuples are immutable.
* Example

**tuple1** = (10, 20, "Apple", 3.4, 'a')

**String :**

* String is a group of characters.
* String values are enclosed either in single quotation or in double quotation marks.
* Example

str1 = 'Hello Friend'  
str2 = "Good Morning"

**Set :**

* Set is an unordered collection of items separated by commas
* The items are enclosed in curly brackets { }.
* A set is similar to list, except that it cannot have duplicate entries.
* Once created, elements of a set cannot be changed.
* Example

set1 = {10,20,3.14,"New Delhi"}

**Dictionary :**

* Dictionary in Python holds data items in key-value pairs.
* Items in a dictionary are enclosed in curly brackets { }.
* Dictionaries permit faster access to data.
* Every key is separated from its value using a colon (:) sign.
* The key: value pairs of a dictionary can be accessed using the  
  key.
* In order to access any value in the dictionary, we have to specify its key in square brackets [ ].
* Example :

dict1 = {'Fruit':'Apple', 'Climate':'Cold', 'Price':120}

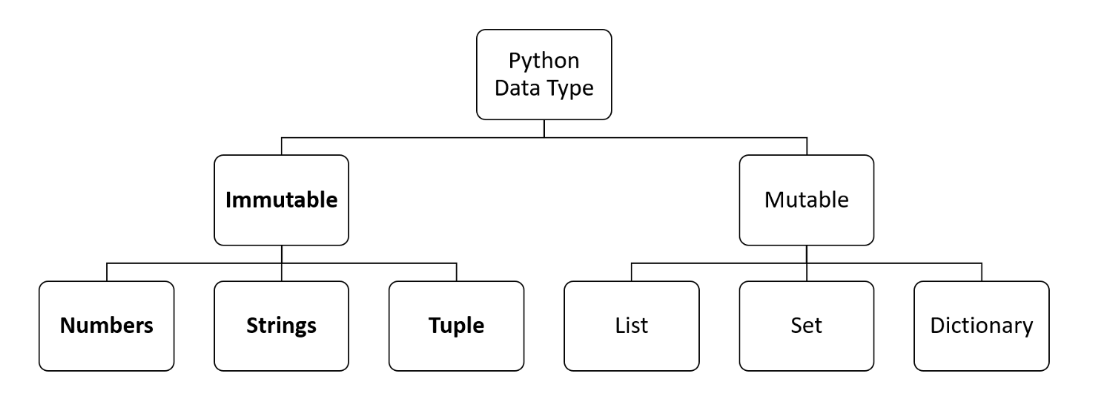
In built function “type” can be used to find type of any object. For ex.

list1 = [5, 3.4, "New Delhi", "20C", 45]  
print(type(list1)) ------------------------------------------ Output will be <class 'list'>

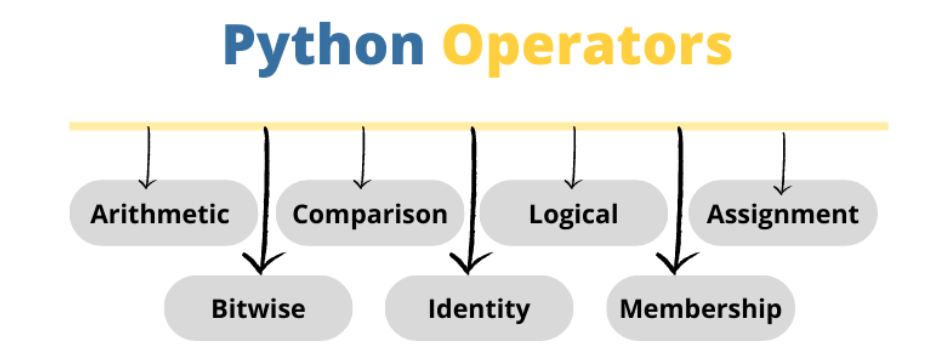
dict1 = {'Fruit':'Apple', 'Climate':'Cold', 'Price':120}  
print(type(dict1)) ----------------------------------------- Output will be <class 'dict'>

**Mutable and Immutable datatypes in Python :**

* Variables whose values can be changed after they  
  are created and assigned are called mutable.
* Variables whose values cannot be changed after they are created  
  and assigned are called immutable .

****

**Chapter 2 : Operators**



**Arithmetic Operators : + , - , \* , / , % , // , \*\***

**% is Modulus operator and returns remainder of operation .**

**// is floor division and returns quotient of operation .**

**\*\* is exponent operator and returns power**

print(32 % 5 ) ------ Outout is 2 ( Remainder of 32 divided by 5)  
print(32 // 5) ------ Output is 6 ( Quotient of 32 divided by 5 )  
print(2\*\*4) ------ Output is 16 ( 2 power 4 is 16)

**Relational Operators :** == , != , > , < , >= , <=

**Assignment Operators : = , + = , - = , \* = , /= , % = , //= , \*\* =**

**Logical Operators : and , or , not**

**Membership Operators : in , not in**

**Chapter 3 : Input and Output**

**Input**

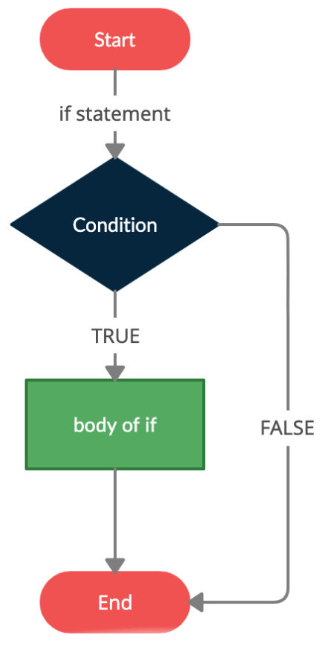
* In Python, we have the input() function for taking the user input.
* The input() function prompts the user to enter data.
* The user may enter a number or a string but the input() function treats  
  them as strings only
* Type conversion is done to convert input data into required type like int() function helps to convert string input into integer

**Output**

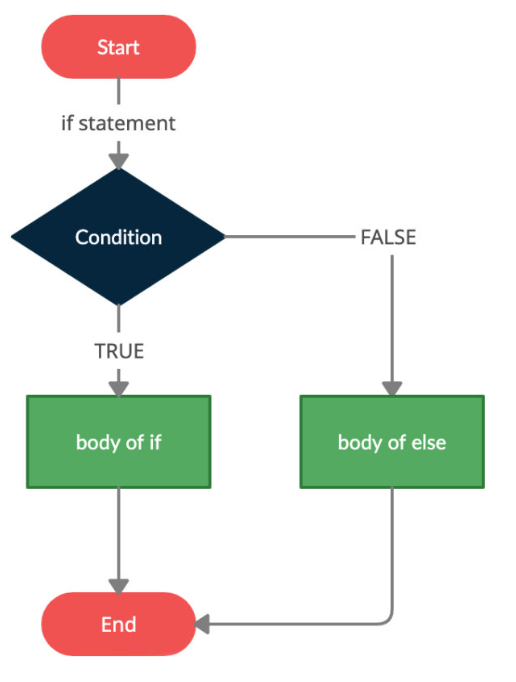
* Python uses the print() function to output data
* function print() evaluates the expression before displaying it.
* The print()outputs a complete line and then moves to the next line
* We can use parameter end = “ “ to avoid output coming in next line

* **Chapter 4 : Decision making ( if -elif-else)**

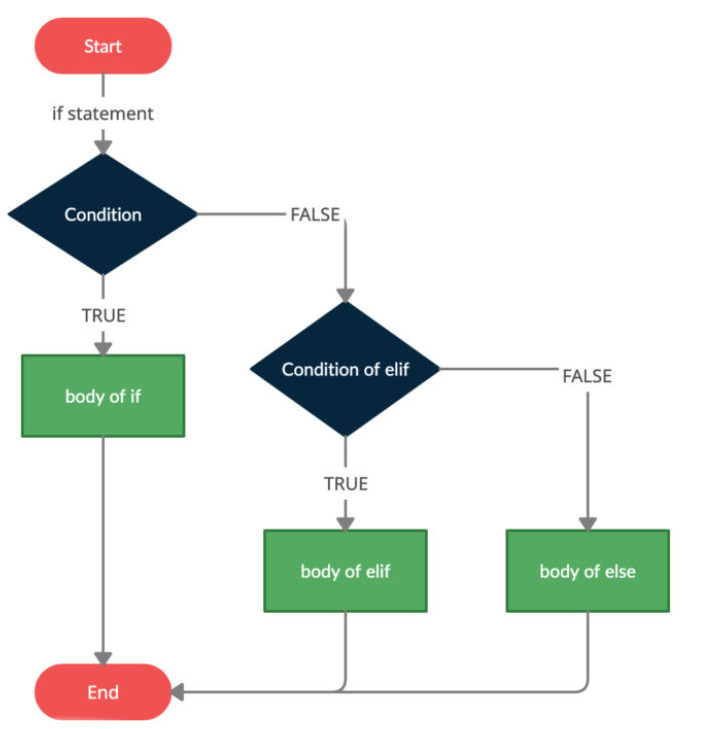
**if :**

****

**if – else :**

****

**if -elif -else :**

****

* In most programming languages, the statements within a block are put inside curly brackets

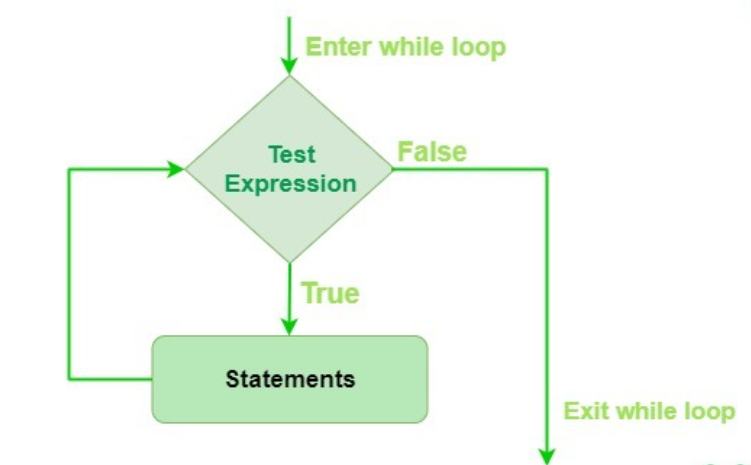
But

Python uses indentation for identifying if/elif/else block

* Leading whitespace (spaces and tabs) at the beginning of a statement is called indentation.
* In Python, the same level of indentation associates statements into a single block of code. It is a common practice to use a single tab for each level of indentation
* **Chapter 5 : Looping ( while/for)**

**Python has 2 looping constructs for repetition : while and for**

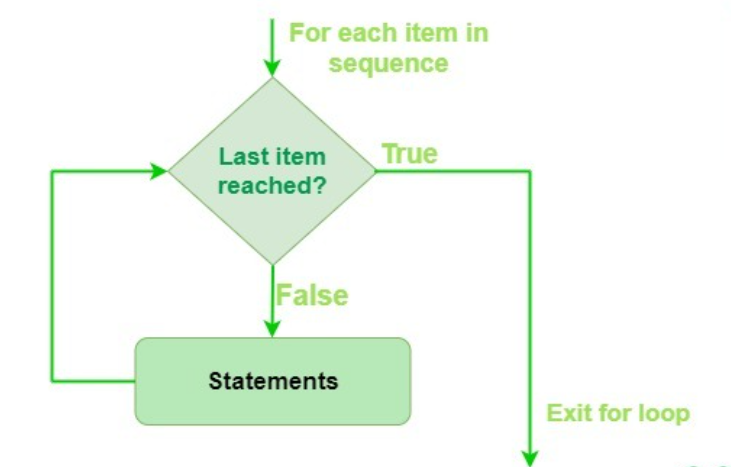
while loop :



count = 0  
**while** count<5:  
 print(**"**Good Morning**"**)  
 count = count + 1

This code print Good Morning 5 times .

for loop :



count = [10,20,30,40,50]  
**for** num **in** count:  
 print(num)

This code will print numbers of the sequence.

**The range() Function** :

* The range() is a built-in function in Python
* Syntax of range() function is:  
  range([start], stop[, step])
* It is used to create a list containing a sequence of integers

from the given start value upto stop value (excluding stop value), with a difference of the given step value .

* range function is usually used a lot in conjuction of for loop.

print(list(range(5))) Output is [0, 1, 2, 3, 4]  
print(list(range(2,5))) Output is [2, 3, 4]  
print(list(range(2,10,3))) Output is [2, 5, 8]

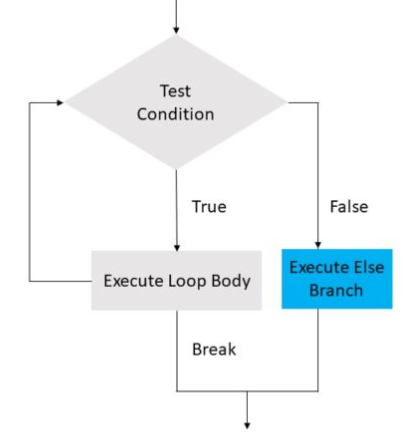
**for** i **in** range(5):  
 print(i,end=**" "**) Output is 0, 1, 2, 3, 4   
   
**for** i **in** range(2,5):  
 print(i,end=**" "**) Output is 2, 3, 4  
   
**for** i **in** range(2,10,3):  
 print(i,end=**" "**) Output is 2, 5, 8

**break statement :**

The break statement terminates the current loop and resumes execution of the statement following that loop

**for** num **in** range(1,10):  
 print(num, end=**" "**) Output is 1 2 3 4 5  
 **if** num == 5:  
 **break**

**for - else :**

****

For example

**for** num **in** [10,20,30,40,50]:  
 **if** num % 2 != 0:  
 print()  
 print(**"Encountering an odd number. Breaking loop"**)  
 **break** print(num,end=**" "**)  
**else**:  
 print()  
 print(**"There is no odd number in list"**)

The output of above code will be

10 20 30 40 50

There is no odd number in list

But if I change list to [10,20,30,41,50]

**for** num **in** [10,20,30,41,50]:  
 **if** num % 2 != 0:  
 print()  
 print(**"Encountering an odd number. Breaking loop"**)  
 **break** print(num,end=**" "**)  
**else**:  
 print()  
 print(**"There is no odd number in list"**)

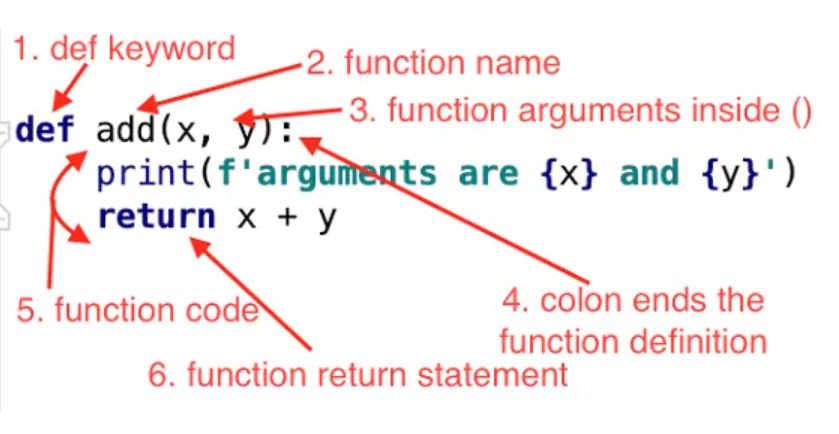
Output will be

10 20 30

Encountering an odd number. Breaking loop

**Chapter 6 : Functions**

* In programming, the use of function is one of the means to achieve modularity and reusability.
* Function can be defined as a named group of instructions that  
  accomplish a specific task when it is invoked.
* Once defined, a function can be called repeatedly from  
  different places of the program without writing all the  
  codes of that function everytime .
* A function is called by simply writing the name of the function and passing the required parameters
* In Python function definition begins with def
* A function may or may not have parameters.
* A function may or may not return a value.
* Function header always ends with a colon (:)
* Function name should be unique. Rules for naming identifiers also applies for function naming
* The statements outside the function indentation are not considered as part of the function.
* input( ) , print( ) , range( ), len( ) etc are in build function

****

**def** add(x,y):  
 print(**f'arguments are {**x**} and {**y**}'**)  
 **return** x+y  
  
  
result = add(10,20)  
print(result) Output 30

**Module:**

* Other than the built-in functions, the Python standard  
  library also consists of a number of modules.
* While a function is a grouping of instructions, a module  
  is a grouping of functions.
* For a complex problem, it may not be feasible to manage the code in one single file. Then, the program is divided into different parts under  
  different levels, called modules.
* A module is created as a python (.py) file containing a collection of function definitions
* To use a module, we need to import the module.
* Once we import a module, we can directly use all the functions of that module.
* The syntax of import statement is as follows:  
  import modulename
* To call a function of a module, the function name should be preceded with the name of the module with a dot(.) as a separator

modulename.functionname()

* Python library has many built-in modules that are really handy to programmers

For ex . math , random etc

**import** math  
  
result = math.sqrt(36)  
print(result) **Output is 6.0 as square root of 36 is 6.0**

* Instead of loading all the functions into memory by importing a module, from statement can be used to access only the required functions from a module. It  
  loads only the specified function(s) instead of all the functions in a module

from modulename import functionname

* To use the function when imported using "from statement" we do not need to precede it with the module name. We can directly call the function

**from** math **import** sqrt  
  
result = sqrt(36)  
print(result) Output is 6.0 as square root of 36 is 6.0

**Chapter 7 : Lists**

* list is an ordered sequence made up of one or more elements
* mutable
* a list can have elements of different data types, such as integer,  
  float, string, tuple or even another list.
* Elements of a list are enclosed in square brackets and are separated by comma
* To access any element in list we need to pass index.
* List index starts from 0
* List has methods like append(), sort() ,reverse() , count() etc .
* In PyCharm by putting a dot after a list object we can see all available methods



You can refer to Python documentation to see usage of list methods

<https://docs.python.org/3/tutorial/datastructures.html>

**Accessing Elements in a List :**

list1 = [2,4,6,8,10,12]  
print(list1[0]) Output is 2  
print(list1[3]) Output is 8  
print(list1[15]) IndexError

print(list1[-1]) Output is 12

**Lists are Mutable :**

list1 = [**'Red'**,**'Green'**,**'Blue'**,**'Orange'**]  
list1[3] = **'Black'**print(list1) Output is ['Red', 'Green', 'Blue', 'Black']

**Membership : in and not in**

list1 = [**'Red'**,**'Green'**,**'Blue'**]  
**if 'Green' in** list1:  
 print(**"Green is there in list"**)  
**else**:  
 print(**"Green is NOT there in list"**)

Output : Green is there in list

list1 = [**'Red'**,**'Green'**,**'Blue'**]  
**if 'White' not in** list1:  
 print(**"White is NOT there in list"**)  
**else**:  
 print(**"Green is there in list"**)

Output : White is NOT there in list

**Slicing :**

* slicing operation is applied to lists to slice list.
* list1 =[**'Red'**,**'Green'**,**'Blue'**,**'Cyan'**,**'Magenta'**,**'Yellow'**,**'Black'**]  
    
  print(list1[2:6]) *# Output ['Blue', 'Cyan', 'Magenta', 'Yellow']*print(list1[:5]) *# Output ['Red', 'Green', 'Blue', 'Cyan', 'Magenta']*print(list1[0:6:2]) *# Output ['Red', 'Blue', 'Magenta']*print(list1[-6:-2]) *# Output ['Green', 'Blue', 'Cyan', 'Magenta']*print(list1[::2]) *# Output ['Red', 'Blue', 'Magenta', 'Black']*print(list1[::-1]) *# Output ['Black', 'Yellow', 'Magenta', 'Cyan', 'Blue', 'Green', 'Red']*

**for loop on a List :**

list1 = [**'Red'**,**'Green'**,**'Blue'**]  
**for** colour **in** list1:  
 print(**f"Colour is {**colour**}"**)

**Output is :**

Colour is Red

Colour is Green

Colour is Blue

**Nested lists :**

Lists inside list is called as nested lists

list1 = [10,20,30,[40,[50,60],70],80,90,100]  
  
print(list1[3]) *# Output is [40, [50, 60], 70]*print(list1[3][1]) *# Output is [50, 60]*print(list1[3][1][0]) *# Output is 50*

**List Comprehension :**

List comprehension is a way of creating lists in very compact manner

squares=[x\*\*2 **for** x **in** range(1,6)]  
print(squares)

Output is [1, 4, 9, 16, 25]

evens = [x **for** x **in** range(1,11) **if** x % 2 == 0]  
print(evens)

Output is [2, 4, 6, 8, 10]

obj = [**"Even" if** i%2 == 0 **else "Odd" for** i **in** range(10)]  
print(obj)

Output is ['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']

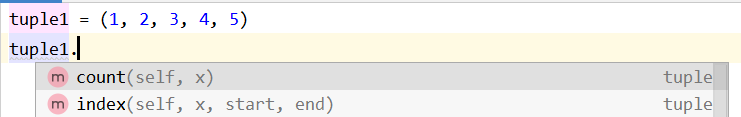
**Chapter 8 : Tuples**

* Like list a tuple is an ordered sequence of elements of different data types.
* Elements of a tuple are enclosed in parenthesis and are separated by commas.
* Like list elements of a tuple can be accessed using index values, starting from 0
* Tuple is an immutable data type

tuple1 = (1, 2, 3, 4, 5)  
tuple1[4] = 10  
print(tuple1)

Output is : TypeError: 'tuple' object does not support item assignment

* On tuple membership,indexing,slicing,for loop works same as list
* Since tuple is immutable it has only read related methods.



**Chapter 9 : Dictionary**

* It is a mapping between a *set of keys* and a *set of values*
* The key-value pair is called an *item*
* A key is separated from its value by a colon(:)
* Items are separated by commas
* Items in dictionaries are unordered, so we may not get back the data in the same order in which we had entered the data initially in the dictionary
* Dictionaries are mutable
* Keys should be unique
* keys should be immutable

Example

dict1 = {**"India"**:**"Delhi"**,**"England"**:**"London"**,**"France"**:**"Paris"**,**"Bangladesh"**:**"Dhaka"**}

**Accessing Items in a Dictionary :**

The items of a dictionary are accessed via the keys

dict1 = {**"India"**:**"Delhi"**,**"England"**:**"London"**,**"France"**:**"Paris"**,**"Bangladesh"**:**"Dhaka"**}  
  
print(dict1[**"France"**]) Output is Paris

print(dict1[**"USA"**]) Output is KeyError: 'USA'

**Adding a new item :**

dict1 = {**"India"**: **"Delhi"**, **"England"**: **"London"**}  
dict1[**"France"**] = **"Paris"**print(dict1)

Output is

{'India': 'Delhi', 'England': 'London', 'France': 'Paris'}

**Modifying an Existing Item :**

dict1 = {**"India"**: **"Delhi"**, **"England"**: **"London"**}  
dict1[**"India"**] = **"New Delhi"**print(dict1)

Output is {'India': 'New Delhi', 'England': 'London'}

**Membership :**

The membership operator in checks if the key is present in the dictionary and returns True, else it returns False

dict1 = {**"India"**: **"Delhi"**, **"England"**: **"London"**}  
  
**if "USA" is** dict1:  
 print(**"Key USA is there"**)  
**else**:  
 print(**"Key USA is NOT there"**)

Output is Key USA is NOT there

**Traversing a dictionary :**

dict1 = {**"India"**: **"Delhi"**, **"England"**: **"London"** , **"France"**:**"Paris"**}  
  
**for** key **in** dict1:  
 print(**f" Capital of {**key**} is {**dict1[key]**}"**)

Output is

Capital of India is Delhi

Capital of England is London

Capital of France is Paris

Another way of traversing dictionary is by using .items( ) method of dictionary

dict1 = {**"India"**: **"Delhi"**, **"England"**: **"London"** , **"France"**:**"Paris"**}  
  
**for** key,value **in** dict1.items():  
 print(**f" Capital of {**key**} is {**value**}"**)

Output is

Capital of India is Delhi

Capital of England is London

Capital of France is Paris

**Dictionary methods :**

You can refer to Python documentation to see usage of list methods

****

Some common methods in dictionary are .keys( ) , .values( ) , .get( ), .update , .items( )

statesAndCapitals1 = {  
 **'Gujarat'**: **'Gandhinagar'**,  
 **'Sikkim'**: **'Gangtok'**,  
 **'Karnataka'**: **'Bangalore'**}  
  
statesAndCapitals2 = {  
 **'Maharasthra'**: **'Mumbai'**,  
 **'Tamil Nadu'**: **'Chennai'**,  
}  
print(list(statesAndCapitals1.keys()))

Output is : ['Gujarat', 'Sikkim', 'Karnataka']

print(list(statesAndCapitals1.values()))

Output is : ['Gandhinagar', 'Gangtok', 'Bangalore']

print(statesAndCapitals1.get(**"Sikkim"**))

Output is Gangtok

statesAndCapitals1 = {  
 **'Gujarat'**: **'Gandhinagar'**,  
 **'Sikkim'**: **'Gangtok'**,  
 **'Karnataka'**: **'Bangalore'**}  
  
statesAndCapitals2 = {  
 **'Maharasthra'**: **'Mumbai'**,  
 **'Tamil Nadu'**: **'Chennai'**,  
}  
  
statesAndCapitals1.update(statesAndCapitals2)  
print(statesAndCapitals1)

Output is :

{'Gujarat': 'Gandhinagar', 'Sikkim': 'Gangtok', 'Karnataka': 'Bangalore', 'Maharasthra': 'Mumbai', 'Tamil Nadu': 'Chennai'}

**for** k,v **in** statesAndCapitals1.items():  
 print(**f"Capital of {**k**} is {**v**}"**)

Output is

Capital of Gujarat is Gandhinagar

Capital of Sikkim is Gangtok

Capital of Karnataka is Bangalore

**Dictionary comprehension** :

Dictionary comprehension allows us to create dictionaries using simple expressions.

square\_dict = {num: num\*\*2 **for** num **in** range(1, 6)}  
print(square\_dict)

Output is

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

new\_dict\_comp = {n:n\*\*2 **for** n **in** range(1,11) **if** n%2 == 0}  
print(new\_dict\_comp)

Output is

{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}

**Chapter 10 : Classes ,Objects ,Methods**

* Object-Oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming
* Everything in Python is really an object
* A class is a special data type which defines how to build a certain kind of object
* The class also stores some data items that are shared by all the instances of this class
* Instances are objects that are created which follow the definition given inside of the class
* Classes have methods that operate on data of the class . Methods are nothing but functions inside class.
* There must be a special first argument self in all of method definitions
* \_\_init\_\_ serves as a constructor for the class. Usually does some initialization work
* The arguments passed to the class name are given to its \_\_init\_\_() method
* An \_\_init\_\_ method can take any number of arguments , however, the first argument self in the definition of \_\_init\_\_ is special
* The first arguments of method,self , is a reference to the current instance of the class . In \_\_init\_\_, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called .
* Python passes “self” automatically on calling methods

**class** student:  
 *# initialisation* **def** \_\_init\_\_(self,n,a):  
 self.full\_name = n  
 self.age = a  
  
 *#getter* **def** get\_age(self):  
 **return** self.age  
  
  
f = student(**"Sachin"**, 20) *# Creating an object and storing its reference*print(f.full\_name) *# Output is Sachin*print(f.age) *# Output is 20*

* The non-method data stored by objects are called attributes
* Data attributes

• Variable owned by a particular instance of a class

• Each instance has its own value for it

* Class attributes
  + Owned by the class as a whole
  + All class instances share the same value for it
  + Because all instances of a class share one copy of a class attribute, when any instance changes it, the value is changed for all instances
* Class attributes are defined within a class definition and outside of any method

Instance variable : In below example ‘x’ is an instance variable

**class** sampleClass:  
  
 **def** \_\_init\_\_(self):  
 self.x = 100  
  
 *# setter* **def** set\_X(self,x):  
 self.x = x  
  
 *# getter* **def** get\_X(self):  
 **return** self.x  
  
 **def** increment\_X(self):  
 self.x = self.x + 1  
  
obj1 = sampleClass()  
obj2 = sampleClass()  
  
print(**f"value of x in obj1 before call to increment is {**obj1.x**}"**)  
print(**f"value of x in obj2 before call to increment is {**obj2.x**}"**)  
  
print(**"#### Calling increment( ) method only for obj1 ####"**)  
obj1.increment\_X()  
  
obj1.get\_X() *# setting x of obj1 as 100*obj2.get\_X() *# setting x of obj2 also to 100*print(**f"value of x in obj1 after call to increment is {**obj1.x**}"**)  
print(**f"value of x in obj2 after call to increment is {**obj2.x**}"**)

Output would be

value of x in obj1 before call to increment is 100

value of x in obj2 before call to increment is 100

#### Calling increment( ) method only for obj1 ####

value of x in obj1 after call to increment is 101

value of x in obj2 after call to increment is 100

So we can see value of x is dfifferent in both objects.

**Class variable :**

In below example x is outside any method hence it’s a class variable

**class** sampleClass:  
 x = 100  
  
 @classmethod  
 **def** increment(cls):  
 cls.x = cls.x + 1  
  
  
obj1 = sampleClass()  
obj2 = sampleClass()  
print(**f"value of x in obj1 is {**obj1.x**}"**)  
print(**f"value of x in obj2 is {**obj2.x**}"**)  
  
print(**"########## Calling increment( ) method only for obj1 "**)  
obj1.increment() *# calling increment( ) method for obj1 method*print(**"########### after increment ###########"**)  
print(**f"value of x in obj1 after call to increment is {**obj1.x**}"**)  
print(**f"value of x in obj2 after call to increment is {**obj2.x**}"**)

Output is

value of x in obj1 is 100

value of x in obj2 is 100

########## Calling increment( ) method only for obj1

########### after increment ###########

value of x in obj1 after call to increment is 101

value of x in obj2 after call to increment is 101

So we see even though the method was called for obj1, the value of x for obj2 also got modified because all instances of a class share one copy of a class attribute, when any instance changes it, the value is changed for all instances.

**staticmethod and classmethod** :

classmethod :

* A class method is a method that is bound to a class rather than itsobject.
* It doesn't require creation of a class instance and can be called directly with classname
* They are used to access or modify the class state
* The class method has a cls parameter which refers to the class.
* Decorator @classmethod is used on top of a method that has to made class method.

Static method :

* static methods are like class methods which can be accessed using class name
* In a static method we don’t have any cls or self parameter.
* They don’t need to (and cannot) access class data
* They should be completely self-contained,and only work with data passed in as arguments
* Static methods are great for utility functions, which perform a task in isolation
* Decorator @staticmethod is used on top of a method that has to made static method

**class** Calculator:  
 *# create addnumbers static method* @staticmethod  
 **def** addnumbers(x, y): # addnumbers methods is not and can’t access “self”  
 **return** x + y  
  
  
print(**'Sum:'**, Calculator.addnumbers(15, 110))

Output : Sum: 125

**Chapter 11 : Exceptions**

* There are various runtime errors like KeyError , AttributeError , IndexError , ZeroDivisionError etc .
* Python keywords related to exception handling are try, except ,finally and raise.
* The block of code ,that can possibly cause exception in kept inside try block .
* except blocks are handlers of exceptions
* finally block always executes after normal termination of try block or after try block terminates due to some exception . Thus finally block is used to deallocate the system resources like file close or sql connection close .finally is optional

**try**:  
 numerator = 50  
 denominator = **"A"** quotient = (numerator / denominator)  
 print(**"Division performed successfully"**) # This line won’t be executed  
**except** KeyError:  
 print(**"Key Not there"**)  
**except** AttributeError:  
 print(**"Attribute Not there"**)  
**except** TypeError:  
 print(**"Incorrect types for operation"**)  
**except** IndexError:  
 print(**"Index Not there"**)  
**except** ZeroDivisionError:  
 print(**"Denominator as ZERO is not allowed"**)  
**else**:  
 print(**"The result of division operation is "**, quotient)  
**finally**:  
 print(**"OVER AND OUT"**)

Output will be :

Incorrect types for operation

OVER AND OUT

* user defined exceptions :
  + Exception is baseclass for all exceptions,so to create our own exception class we need to extend Exception
  + Built-in exceptions are raised automatically but keyword "raise" is used to raise user defined exception

**class** MyException(Exception):  
 **def** \_\_init\_\_(self,msg):  
 self.msg=msg  
  
  
x = 101  
**try**:  
 **if** x>100:  
 **raise** MyException(**"max marks can be 100 only"**)  
**except** TypeError:  
 print(**"Incorrect types for operation"**)  
**except** IndexError:  
 print(**"Index Not there"**)  
**except** MyException **as** obj:  
 print(obj.msg)  
**except**:  
 print(**"Some Error happened"**)  
**finally**:  
 print(**"i m in finally"**)

Output will be :

max marks can be 100 only

i m in finally

**Chapter 12 : Function arguments**

* Positional arguments

**def** add(a,b,c):  
 sum = a+b+c  
 **return** sum  
  
  
result = add(10,20,30)  
print(result) Output 60

a,b,c are positional arguments

* Default arguments

**def** add(a,b,c=100):  
 sum = a+b+c  
 **return** sum  
  
  
result = add(10,20)  
print(result)

c is by default 100 . So if value is not passed for c default 100 will be taken and if value is passed that value will be taken .

* Keyword arguments : values are sent along keywords

**def** profile(name,sports,country):  
 print(**f"{**name**} plays {**sports**} for {**country**}"**)  
  
  
profile(sports =**"Badminton"**,country= **"India"**,name=**"Lakshya Sen"**)  
profile(country= **"India"**,name=**"Rohit Sharma"**,sports =**"Cricket"**)

Output would be :

Lakshya Sen plays Badminton for India

Rohit Sharma plays Cricket for India

* Variable argurments (\*args)

\*args allow us to pass the variable number of non keyword arguments to function

In the function, we should use an asterisk \* before the parameter name to pass variable length arguments.The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk

**def** add(\*args):  
 sum = 0  
 **for** num **in** args:  
 sum = sum + num  
 print(**"Sum:"**, sum)  
  
add(3, 5)  
add(4, 5, 6, 7)  
add(1, 2, 3, 5, 6)

Output is

Sum: 8

Sum: 22

Sum: 17

* Variable keyword arguments

\*\*kwargs, it allows us to pass the variable length of keyword arguments to the function.

In the function, we use the double asterisk \*\* before the parameter name to denote this type of argument. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk \*\*.

**def** profile(\*\*kwargs):  
 **for** k,v **in** kwargs.items():  
 print(**f"{**k**} = {**v**}"**)  
  
  
profile(name = **"Lakshya"**)  
print(**"###"**)  
profile(name = **"Lakshya"**,sports=**"Badminton"**)

Output is

name = Lakshya

###

name = Lakshya

sports = Badminton

**Chapter 13 : Inheritance**

* Inheritance is OOP concept that enables us to define a class that takes all the functionality from a parent class and allows us to add more

Syntax to inherit a class

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

**class** Grandfather:  
 **def** add(self,a,b):  
 **return** a+b  
  
**class** Father(Grandfather):  
 **def** product(self,a,b):  
 **return** a\*b  
  
**class** Son(Father):  
 **def** power(self,a,b):  
 **return** a\*\*b  
  
  
child = Son()  
print(child.add(2,5))  
print(child.product(2,5))  
print(child.power(2,5))

So we see child object is an object of Son class and it has automatically got access to all methods of parent and grandparent.

Output is :

7

10

32

**Chapter 14 : Decorators**

A decorator is a function that takes in a function as argument and returns a function as argument.

Decorators are used to modify functionality of a function without modifying that function .

In Python everything is object . If function is an object .

def first(msg):

print(msg)

first("Hello")

second = first

second("Hello")

Here, the names first and second refer to the same function object.

Functions can be passed as arguments to another function.

**def** incrementbyone(x):  
 **return** x + 1  
  
  
**def** decrementbyone(x):  
 **return** x - 1  
  
  
**def** operate(func, x):  
 result = func(x)  
 **return** result  
  
print(operate(incrementbyone,10)) *# function reference is passed*print(operate(decrementbyone,10)) *# function reference is passed*

Output is

11

9

A function can return another function.

**def** outer():  
 **def** inner():  
 print(**"Hello World"**)  
 **return** inner  
  
  
new = outer()  
new() *# Now this is equivalent to inner()*

**def** decorator(func):  
 **def** inner():  
 print(**"\*\*\*\*\*\*"**)  
 func()  
 print(**"\*\*\*\*\*\*"**)  
 **return** inner  
  
  
**def** ordinary():  
 print(**"Good Morning"**)  
  
ordinary = decorator(ordinary)  
ordinary()

Output is

\*\*\*\*\*\*

Good Morning

\*\*\*\*\*\*

Here firstly the decorator fucntion( decorator) is called and the function to be decorated ( ordinary ) is passed as argument .

This means func = ordinary which means func refers to ordinary function object now .

The decorator function returns reference of function inner

( return inner )

inner is actually wrapper to ordinary function .

The reference returned by decorator (inner) is assigned to name of the function to be decorated . This means

ordinary = inner

Now finally ordinary is called and since ordinary = inner , this means actually inner function will be called .

And since func = ordinary thus ordinary function is called from inner function.

So the sequence is

func = ordinary

ordinary = inner

So call to ordinary (ordinary( ) ) now means call to inner( ) which will internally call func ( ) and as func = ordinary , ordinary function will be called from inner function, thus doing the decoration job.

Python provide @ symbol to call decorator directly without calling decorator function explicitly

**def** decorator(func):  
 **def** inner():  
 print(**"\*\*\*\*\*\*"**)  
 func()  
 print(**"\*\*\*\*\*\*"**)  
 **return** inner  
  
@decorator  
**def** ordinary():  
 print(**"****Good Morning"**)  
  
ordinary()

If we remove @decorator from top of ordinary() output of this code will be

Good Morning.

But if we put @decorator on top of ordinary( ) output will be

\*\*\*\*\*\*

Good Morning

\*\*\*\*\*\*

Another example

**def** decor(fun):  
 **def** inner():  
 value = fun()  
 **return** value + 2  
 **return** inner  
  
@decor  
**def** num():  
 **return** 10  
  
print(num())

If we remove @decor output of this code will be 10 .

But if we put @decor output of this code wii be be 12.

Thus decorator is modifying functionality of function without modifying it.

**Chapter 15 : lambda function**

Also called as anonymous function

It is a function without name

anonymous functions are defined using the lambda keyword.

Syntax of lambda function is

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example

double = lambda x: x \* 2

print(double(5))

Output is 10

lambda is commonly used with inbuild function map() , filter() and reduce()

Syntax of map is

map(func,sequence)

map function applies function func on each element of sequence. For single expression function instead of a normal function a lambda function can be used .

Example

mylist =[2,3,5,7,8]  
result = map(**lambda** x:x\*\*2 , mylist)  
print(list(result))

Output is [4, 9, 25, 49, 64]

* Like map syntax of filter also is

filter(func,sequence)

but the function (func) that does job of filtering returns only True or False. filter function applies filtering job on sequence based logic of func.

mylist =[2,3,5,7,8]  
result = filter(**lambda** x:**True if** x%2==0 **else False** , mylist)  
print(list(result))

Output is [2, 8] . Here the lambda function is helping to find even numbers of the sequence .

* Like map and filter syntax of reduce is also

reduce(func,sequence)

reduce helps to aggregate sequence to yield a single output dpeneding on logic of function func.

**from** functools **import** reduce  
mylist =[2,3,5,7,8]  
result = reduce(**lambda** x,y:x+y, mylist)  
print(result)

Output is 25 which is sum of all elements of mylist.