Python

Python is an interpreted, high level, general purpose, and dynamic language.

Python's History

In February 1991python was published by Guido Van Rossum and python's 1st version i.e v.1.0 was launched in Jan1994.

1. The idea of python started in the early 1980s.
2. Real implementation started in 1989 by Guido Van Rossum CWI (Central Wiskunde & Informatica) in Netherlands
3. Finally published on 27 Feb 1991.
4. Officially the first version 1.0 came in 1994.

Features of Python

1. Easy language
2. Readable
3. Interpreted Language
4. Dynamically typed Language
5. Object Oriented
6. Popular and large community Support
7. Open- Source
8. Large standard library
9. Platform- Independent
10. Extensible and Embeddable
11. GUI Support
12. High Level language

Note:- Python is Dynamically typed Language- When assigning to a variable, we do not need to declare the data type of the values it will hold. This is decided by the interpreter at runtime.

Pros and Cons of python

Pros	Cons
Easy to learn and write	Slow speed
Improved productivity	Not memory Efficient
Interpreted Language	Weak in mobile computing
Dynamically typed	Database Access
Free and open Source, Vast Libraries Support	Runtime errors

Application of Python

1. Web development
2. Desktop GUI application
3. Scientific and Numeric Calculation
4. Data Science, ML, AI, Robotics
5. Software Development
6. Business Application
7. Games and 3D Graphics
8. Network Programming
9. Database Access

Variables and Keywords

Variables are nothing, it is just a name which store data. It acts like a container which hold values.

```
E.g.- name='python'

print(name)
```

Here 'name' is variable.

Variables Naming Convention

1. Variables can have letters (A-Z & a-z), digits(0-9), and underscore(_).
2. It cannot have whitespace and sign like +,&,-,!,@,$,#,%
3. It must not Start with number.
4. It cannot be a reserved Keyword for Python.
5. Variables name are case sensitive.
6. Variables are alphanumeric.

   E.g- PYTHON ☑

   PYTHON- PROGRAMMING ☒

   PYTHON_PROGRAMMING ☑

   1_PYTHON ☒

   1PYTHON ☒

   PY THON ☒

Python Reserved Keywords

Python keywords are special reserved words that have specific meanings and purpose and cannot be used for anything but those specific purpose. Python keywords are different from python's built in function and types.

Note:- There are 35 reserved keywords in python

Identifier in Python

An Identifier is a name given to an entity. In very simple words, an identifier is a user defined name to represent the basic building blocks of Python. It can be a variables, a functions, a class, a module or any other object.

Naming rules for Identifiers

1. The Python identifiers is made with a combination of lowercase letter(a-z), uppercase letter(A-Z), digits(0-9) or an underscore(_).
2. An identifiers cannot start with a digit.
3. We also cannot use special symbols in the identifier name. Symbols like (!,@,#,$,%)
4. A keyword cannot be used as an identifier. In python Keywords are the reserved names that are inbuilt in python. They have special meaning and we cannot use them as identifier names.

Operators

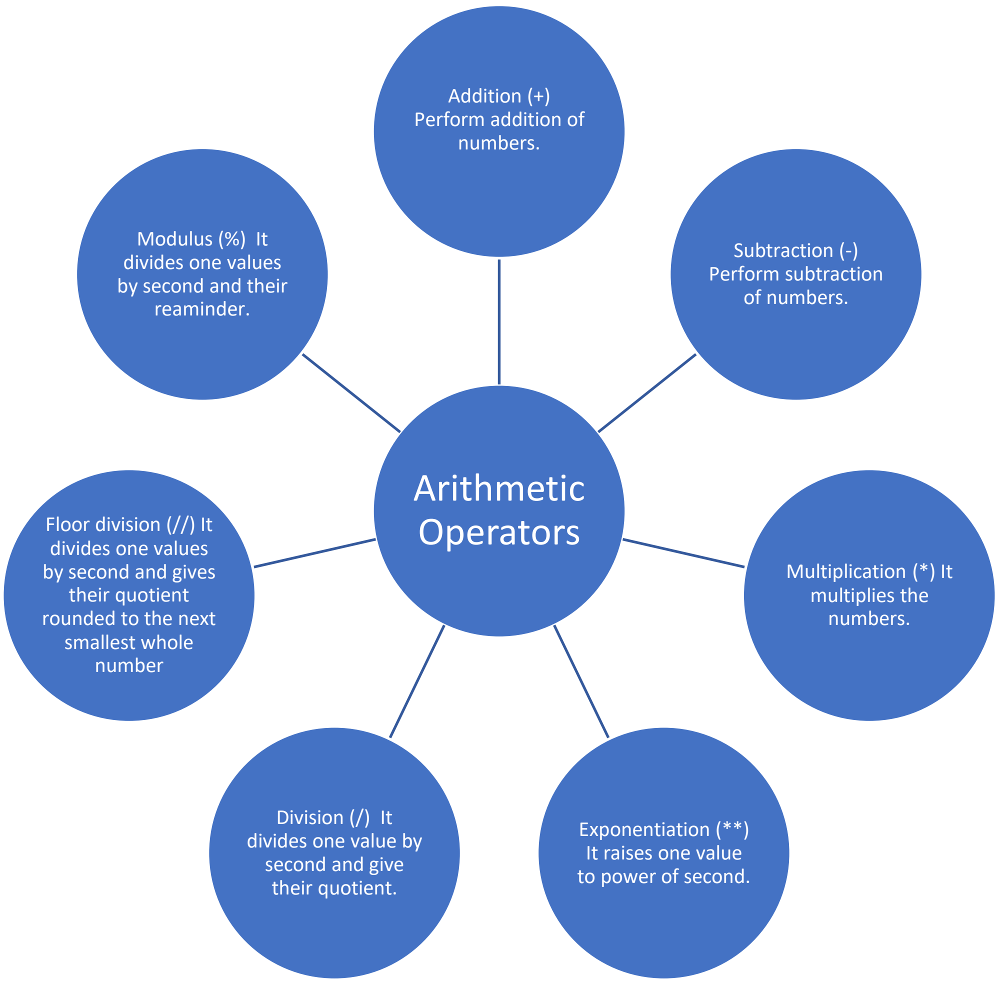
An operator is a symbol that will perform mathematical operations on variables or on values. operators operate on the operands (value) and return a result.

e.g   A+B
Here A and B is working as operand and '+' is working as operator.

Python has 7 types of operators

1. Arithmetic Operators
2. Relational Operators
3. Assignment Operators
4. Logical Operators
5. Membership Operators
6. Identity Operators
7. Bitwise Operators

1. Arithmetic Operators


Arithmetic Operators
Addition (+) Perform addition of numbers.
Subtraction (-) Perform subtraction of numbers.
Multiplication (*) It multiplies the numbers.
Exponentiation (**) It raises one value to power of second.
Division (/) It divides one value by second and give their quotient.
Floor division (//) It divides one values by second and gives their quotient rounded to the next smallest whole number
Modulus (%) It divides one values by second and their reaminder.

Some Examples based on Arithmetic operators

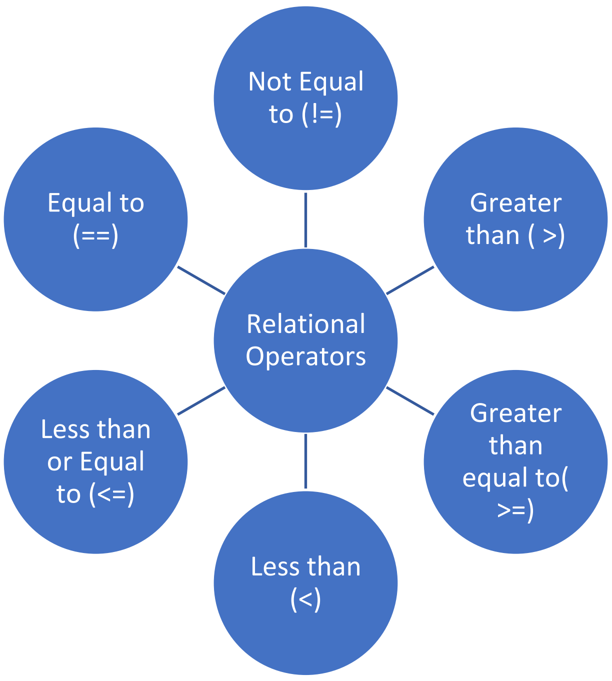
```
x=12
y=24
print('addition',x+y)
print('Subtraction',x-y)
print('Mutliplication',x*y)
print('Exponential',x**y)
print('Division',x/y)
print ('Floor division',x//y)
```

```
print('Modulus',x%y)
```

Output:-

```
addition 36
Subtraction -12
Mutliplication 288
Exponential 79496847203390844133441536
Division 0.5
Floor division 0
Modulus 12
```

2. Relational Operator



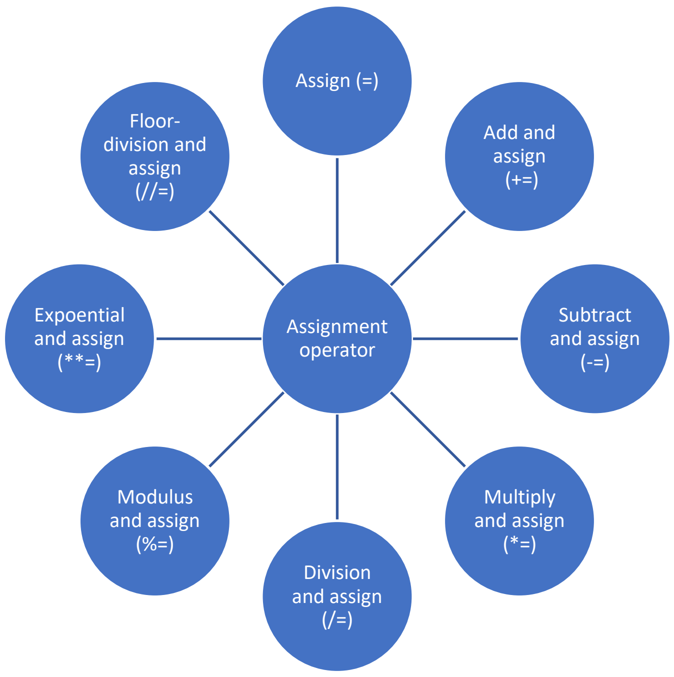
Some Examples based on Arithmetic operators

```
x=12
y=24
print(x>y)
print(x<y)
print(type(x>y))
```

Output:-

```
False
True
<class 'bool'>
```

3. Assignment Operator

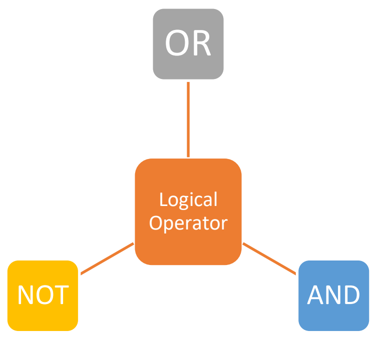
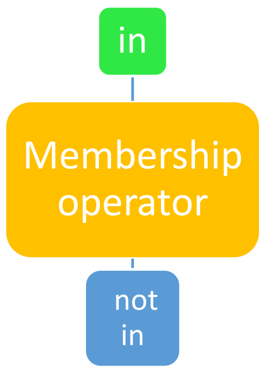
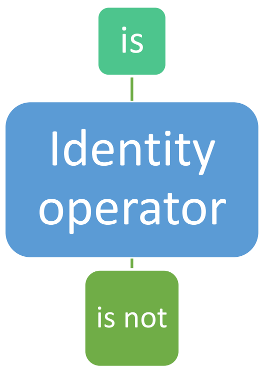


Some Examples based on Assignment operators

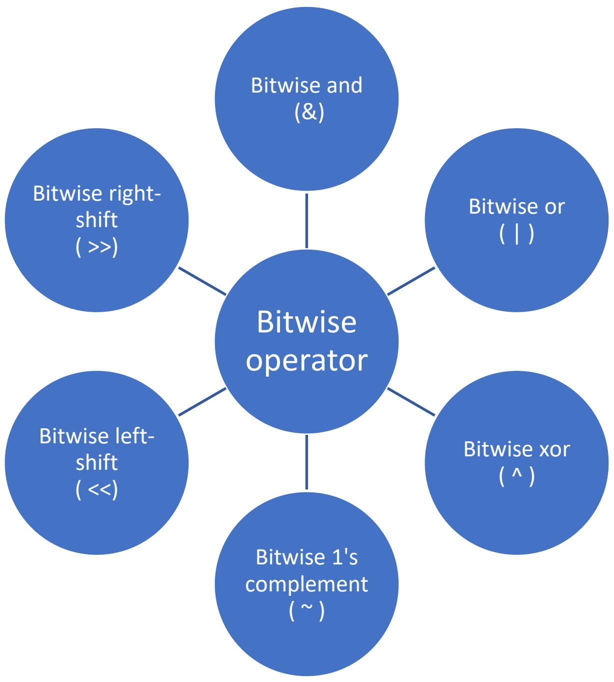
```
x=12
y=24
x+=y
print('add and assign',x)
x-=y
print('Subtract',x)
x*=y
print('Mutliply and assign',x)
x/=y
print('Division and assign',x)
x**=y
print('Exponential and assign',x)
x%=y
print('Modulus and assign',x)
x//=y
print('Floor division and assign',x)
```

Output: -

```
add and assign 36
Subtract 12
Mutliply and assign 288
Division and assign 12.0
Exponential and assign 7.949684720339084e+25
Modulus and assign 0.0
Floor division and assign 0.0
```

4. Logical Operators	5. Membership Operator	6. Identity Operator
		

7.Python Bitwise Operator



Some Examples based on Bitwise operators

```
x=12   # 12=00001100
y= 24  # 24=00011000
z=x&y
print('Bitwise and',z)
z=x|y
print('Bitwise or',z)
z=x^y
print('Bitwise xor',z)
z=~x
print('Bitwise 1s complement', z)
z=x<<2
print('Bitwise left wise shift',z)
```

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

Output:-

```
Bitwise and 8
Bitwise or 28
Bitwise xor 20
Bitwise 1s complement -13
Bitwise left wise shift 48
Bitwise right wise shift 3
```

Ternary Operator

The ternary operator in python is nothing but a one line version of the if-else statement.

Syntax of ternary operator :-
<’true_value’>if <conditional_expression>else<’false_value’>

Example

```
a=12

b=2
'even_number' if a%b==0 else 'odd number'
```

Output: -
even_number

Expression in Python

An expression is made with a combination of variables, operators, values and function codes.

Flow Control

If Statement

If statements is the most simple form of decision -making statement. It takes an expression if condition is true then the block of code inside the if statement will be executed.

E.g
```
 n=int(input('Enter the number:'))
if n>0:
  print('Positive number')
```

Output:-
```
Enter the number:12
Positive number
```

If-else Statement

If else statement check the condition if the condition is true then the code of true condition will be executed otherwise false.

Example
```
n=int(input('Enter the number'))
if n>18:
  print('eligible for vote')
else:
  print('Not eligible')
```
Output:-
```
Enter the number:
```

If-elif Ladder

In python we have elif keyword to chain multiple condition one after the other. With the help of elif ladder we can make complex decision-making statement. The elif statement help to check multiple statement and the give result when the condition is true.

Example:
```
Choice=int(input('Enter the number:'))
print('Select your ride')
if Choice==1:
  print('Bike')
elif (Choice==2):
  print('scooter')
elif(Choice==3):
  print('car')
elif(Choice==4):
  print('mercedes')
else:
  print('none ')
```
Output:-
```
Enter the number:1
Select your ride
Bike
```

Nested if Statement

Nested if statement is an if statement inside another statement. Python allows us to stock any number of if statement inside the block of another if statement inside if statement. They they are useful when we need to make a series of the decision. They are useful when we need to make a series of decision.

Example:-

```
num1=int(input('Enter the number:'))
num2=int(input('Enter the number:'))
if num1>=num2:
  if num1==num2:
    print(f'{num1} is equal to {num2}')
  else:
    print(f'{num1} is greater than {num2}')
else:
  print(f'{num1} is less than {num2}')
```

Python Loop

Loops are most powerful and basic concept in programming. A loop can contain a set of statements that keep on executing until a specific condition is reached.

For loop

For loop in python is used to iterate over a sequence of item like list, tuple, set, dictionary, string or any other iterable object. Int object is not iterable.

Example:-

```
for item in [1,2,3,4]:
  print(item)
```

Output

```
1
2
3
4
```

The range Function.

When using for loop in python, the range() function is pretty useful to specify the number of times the loop is executed. It yields a sequence of numbers within a specified range.

Syntax: -
range(start,stop,step)

Example: -

```
for item in range(1,10,2):
  print(item)
```

Output:-

```
1
3
5
7
9
```

Some more example :-

```python
for i in range(1,5):
  for j in range(1,5):
    print(i,j)
```

 Output:-

```
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
```

```python
rows = int(input('enter number of rows'))

for i in range(1,rows+1):
  for j in range(1,i+1):
    print('*',end='')
  print()
```

Output:-

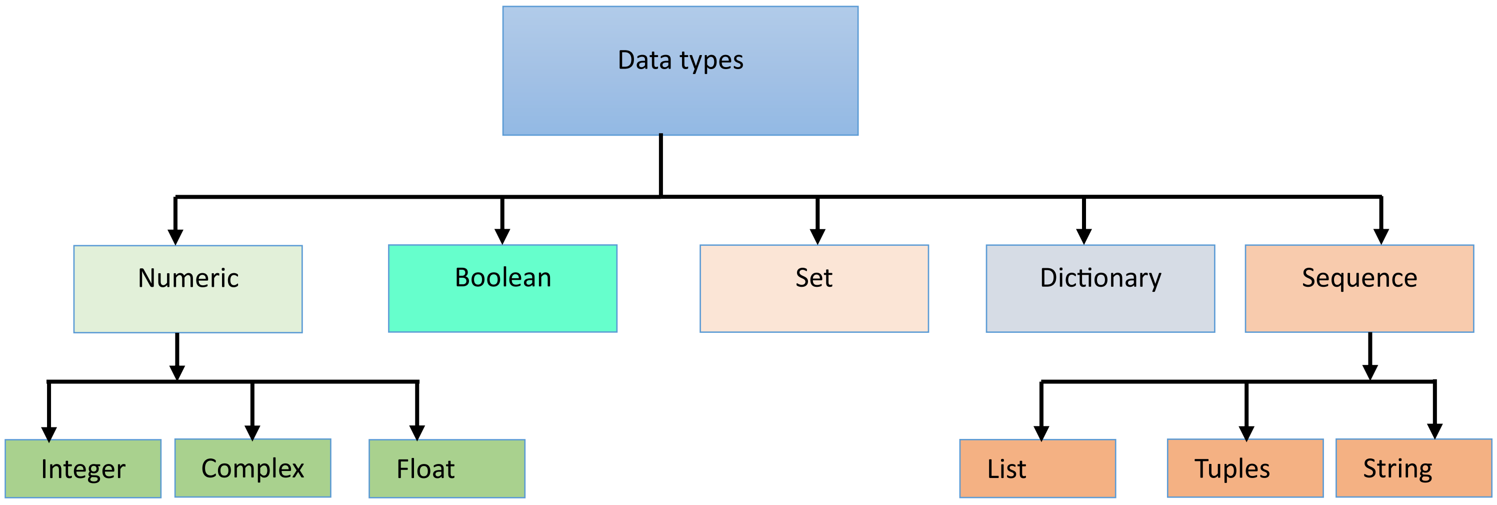
```
*
**
***
****
*****
******
*******
********
*********
**********
```

```python
rows = int(input('enter number of rows'))

for i in range(1,rows+1):
  for j in range(1,i+1):
    print(j,end='')
  for k in range(i-1,0,-1):
    print(k,end='')

  print()
```

Output:-

```
1
121
12321
1234321
```

Datatypes



Numeric Data types

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number.

Integers:- This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.

Floats:- This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point.

Complex Numbers:- Complex number is represented by a complex class.

Examples:-

```
n=1
print(type(n))
n=1.2
print(type(n))
n=1+3j
print(type(n))
```

Output-

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

Boolean

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class bool.

Examples: -

```
print(type(True))
print(type(False))
n=1
m=2
n>m
```

```
print(type(n>m))
```

Output:-

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

Note:- True and False with capital ‘T’ and ‘F’ are valid booleans otherwise python will throw an error.

```
print(type(true))
```

```
Output: -

NameError                                 Traceback (most recent call
last)
<ipython-input-19-2fc8230f6ecf> in <cell line: 29>()
     27 n>m
     28 print(type(n>m))
---> 29 print(type(true))

NameError: name 'true' is not defined.
```

Dictionary

Dictionaries in python are collection that are unordered indexed and mutable. They hold keys and values.

Keys in dictionaries are unique and immutable.

Example:-

```
dict={'car':'mercedes','fruits':'orange'}
print(dict)
print(type(dict))
```

Output:-

```
{'car': 'mercedes', 'fruits': 'orange'}
<class 'dict'>
```

Accessing items in python Dictionaries.

For accessing item in dictionaries we have two methods.
1.By indexing
2.By get() method

1. Indexing- Dictionaries are unordered but we can idea them with their keys.

Example:

```
dict={'car':'mercedes','fruits':'orange'}
print(dict)
print(type(dict))
dict['car'] #Indexing
```

Output:-

```
{'car': 'mercedes', 'fruits': 'orange'}
<class 'dict'>
'
'mercedes'
```

2. get() method

The get() method does the same things as indexing. It takes a parameter key to find its values.

Example:-

```
dict={'car':'mercedes','fruits':'orange'}
print(dict)
print(type(dict))
dict.get('car')
```

Outputs:-

```
{'car': 'mercedes', 'fruits': 'orange'}
<class 'dict'>
'mercedes'
```

Methods	Description
dict.popitems()	Removes the last inserted key-value pair.
dict.pop()	Remove the element with specified key.
dict.items()	Returns a list containing a tuple for each key value pair.
dict.keys()	Returns a list containing dictionary's keys.
dict.update()	Updates dictionary with specified key-value pairs.
dict.values()	Returns a list of all the values of dictionary.
dict.copy()	Returns a copy of the dictionary
dic.clear()	Remove all the elements from the dictionary
dict.get(key)	Returns the value of specified key
dict.setdefault(key,value)	set the key to the default value if the key is not specified in the dictionary
dict.[key]=(values)	Add key and values to the dictionary.
dict.fromkeys(iterable,values)	Iterable can be list, tuple, range(), dictionary

Examples:-

```
dict={'car':'mercedes','fruits':'orange'}
print(dict)
print(type(dict))
dict['car']
dict['fruits']
dict['entertainment']=('games')
print(dict)
dict.get('car')
dict.popitem()
print(dict)
dict.setdefault('dress','suits')
print(dict)
```

```
dict.update()
print(dict)
dict.pop('car')
print(dict)
dict.update()
print(dict)
l=[1,2,34]
dict.fromkeys(l,0)
dict.keys()
dict.values()
for key,values in dict.items():
  print(key)
  print(values)

d1={'1': 'car','2': 'Trucks',  ### Nested Dictionary
    '3':{'A':'Welcome','B':'To','C': 'Home'}}
print(d1)
print(d1['2'])
print(d1['3']['A'])
print(type(d1))
```

```
Output:-
{'car': 'mercedes', 'fruits': 'orange'}
<class 'dict'>
{'car': 'mercedes', 'fruits': 'orange', 'entertainment': 'games'}
{'car': 'mercedes', 'fruits': 'orange'}
{'car': 'mercedes', 'fruits': 'orange', 'dress': 'suits'}
{'car': 'mercedes', 'fruits': 'orange', 'dress': 'suits'}
{'fruits': 'orange', 'dress': 'suits'}
{'fruits': 'orange', 'dress': 'suits'}
{1: 0, 2: 0, 34: 0}
dict_keys(['fruits', 'dress'])
dict_values(['orange', 'suits'])
fruits
orange
dress
suits
{'1': 'car', '2': 'Trucks', '3': {'A': 'Welcome', 'B': 'To', 'C':
'Home'}}
Trucks
Welcome
<class 'dict'>
```

Sequence Data type

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion.

1.List
2.Tuples
3.String

1. List

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.
List are mutable collection of objects. List are heterogeneous. A list can contain different types of element.

1.1 Indexing in list.

Positive indexing starts from left to right.
Negative indexing starts from right to left.

1.2 Slicing in List

We can slice and this gives us only a part of the list.
Syntax:- list=[start: stop: step]

Examples: -

```
list=[11,32,'car','Automobiles','Industry']
print(list)
print(type(list))
list=[11,32,'car','Automobiles','Industry',['1',2,3,['A','B','C']]]  #N
ested List
print(list)
list1=[list,11,32,'car','Automobiles','Industry']   #Nested List
print(list1)
print(type(list1))
print(list[1])  # Indexing in list
print(list[5])  # Indexing in list
print(list1[5]) # Indexing in list
print(list1[-5]) # Indexing in list
print(list1[-1])  # Indexing in list
print(list[:])             # return complete list
print(list[:2])             #return upto index1
print(list[2:])             #return from index2 to end
```

Output: -

```
[11, 32, 'car', 'Automobiles', 'Industry']
<class 'list'>
[11, 32, 'car', 'Automobiles', 'Industry', ['1', 2, 3, ['A', 'B',
'C']]]
[[11, 32, 'car', 'Automobiles', 'Industry', ['1', 2, 3, ['A', 'B',
'C']]], 11, 32, 'car', 'Automobiles', 'Industry']
<class 'list'>
32
['1', 2, 3, ['A', 'B', 'C']]
Industry
11
Industry
[11, 32, 'car', 'Automobiles', 'Industry', ['1', 2, 3, ['A', 'B',
'C']]]
[11, 32]
['car', 'Automobiles', 'Industry', ['1', 2, 3, ['A', 'B', 'C']]]
```

Method	Description
Append()	Add an element to the end of the list
Extend()	Add all elements of a list to another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of the number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
Copy()	Returns a copy of the list
Pop()	Removes and returns the item at the specified index. If no index is provided, it removes and returns the last item.

Example:-

```
list=[11,32,'car','Automobiles','Industry']
print(list)
list.append(5)
print(list)
list.copy()
list.extend('research')
list.count('car')
print(list.count('car'))
print(list)
list.extend('11')
print(list)
list.count('r')
print(list.count('r'))
list.index('car')
print(list.index('car'))
list.insert(2,'Refurbished')
print(list)
list.remove('r')
print(list)
list.pop(0)
print(list)
list.remove('r')
print(list)
list.reverse()
print(list)
list.clear()
print(list)
```

Output-

```
[11, 32, 'car', 'Automobiles', 'Industry']
[11, 32, 'car', 'Automobiles', 'Industry', 5]
```

```
1
[11, 32, 'car', 'Automobiles', 'Industry', 5, 'r', 'e', 's', 'e', 'a',
'r', 'c', 'h']
[11, 32, 'car', 'Automobiles', 'Industry', 5, 'r', 'e', 's', 'e', 'a',
'r', 'c', 'h', '1', '1']
2
2
[11, 32, 'Refurbished', 'car', 'Automobiles', 'Industry', 5, 'r', 'e',
's', 'e', 'a', 'r', 'c', 'h', '1', '1']
[11, 32, 'Refurbished', 'car', 'Automobiles', 'Industry', 5, 'e', 's',
'e', 'a', 'r', 'c', 'h', '1', '1']
[32, 'Refurbished', 'car', 'Automobiles', 'Industry', 5, 'e', 's', 'e',
'a', 'r', 'c', 'h', '1', '1']
[32, 'Refurbished', 'car', 'Automobiles', 'Industry', 5, 'e', 's', 'e',
'a', 'c', 'h', '1', '1']
['1', '1', 'h', 'c', 'a', 'e', 's', 'e', 5, 'Industry', 'Automobiles',
'car', 'Refurbished', 32]
[]
```

List Comprehension

List comprehension is an elegant way to define and create a list in python. We can create lists just like mathematical statements and in one line only.

Examples:-

```
l1=[2,4,7,8,9]
even=[i for i in l1 if i %2==0 ]
print(even)
```

Output:-

```
[2, 4, 8]
```

Tuples

Tuples are collection of python objects. They are similar to lists but the difference between them is list is mutable and tuples are immutable. Tuples are created by typing a sequence of item separated by commas. Optionally, we can put the comma-separated values in parenthesis.

Examples:-

```
 tuples=(1,3,4,'car','Automobiles','car')
 print(tuples)
 print(type(tuples))
 tuples.count('car')
 print(tuples.count('car'))
 tuples.index('car')
 print( tuples.index('car') )
tuples[0]
print(tuples[0])
```

Output-

```
(1, 3, 4, 'car', 'Automobiles', 'car')
<class 'tuple'>
2
3
1
```

String

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once we have created a string, we cannot change it. Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

Creating String in Python

Strings in Python can be created using single quotes or double quotes or even triple quotes. The triple quotes can be used to declare multiline strings in Python.

Accessing characters in Python String

In Python, individual characters of a String can be accessed by using the method of Indexing. While accessing an index out of the range will cause an IndexError. Only Integers are allowed to be passed as an index, float or other types that will cause a TypeError.

String Slicing

In Python, the String Slicing method is used to access a range of characters in the String. Slicing in a String is done by using a Slicing operator, i.e., a colon (:). Example- The [3:12] indicates that the string slicing will start from the 3rd index of the string to the 12th index. [::-1] we can reverse the string by using [::-1], We did not specify the first two parts of the slice indicating that we are considering the whole string, from the start index to the last index.

Methods	Description
string.ascii_letters	Concatenation of the ascii_lowercase and ascii_uppercase constants.
string.ascii_lowercase	Concatenation of lowercase letters
string.ascii_uppercase	Concatenation of uppercase letters
string.digits	Digit in strings
string.hexdigits	Hexadigit in strings
string.letters	concatenation of the strings lowercase and uppercase.
string.lowercase	A string must contain lowercase letters.
String.endswith()	Returns True if a string ends with the given suffix otherwise returns False
String.startswith()	Returns True if a string starts with the given prefix otherwise returns False
String.isdigit()	Returns “True” if all characters in the string are digits, Otherwise, It returns “False”.
String.isalpha()	Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”.
string.isdecimal()	Returns true if all characters in a string are decimal.

str.format()	one of the string formatting methods in Python3, which allows multiple substitutions and value formatting.
String.index	Returns the position of the first occurrence of substring in a string.
string.uppercase	A string must contain uppercase letters.
string.swapcase()	Method converts all uppercase characters to lowercase and vice versa of the given string, and returns it
replace()	returns a copy of the string where all occurrences of a substring is replaced with another substring.
string.Isdecimal	Returns true if all characters in a string are decimal
String.Isalnum	Returns true if all the characters in a given string are alphanumeric.
string.Istitle	Returns True if the string is a title cased string
String.partition	splits the string at the first occurrence of the separator and returns a tuple.
String.Isidentifier	Check whether a string is a valid identifier or not.
String.len	Returns the no of string including the space..
String.Max	Returns the highest alphabetical character in a string.
String.min	Returns the minimum alphabetical character in a string.
string.capitalize	Return a word with its first character capitalized.
string.expandtabs	Expand tabs in a string replacing them by one or more spaces.
string.find	Return the lowest indexing a sub string.
string.count	Return the number of (non-overlapping) occurrences of substring sub in string.
string.lower	Return a copy of string, but with upper case, letters converted to lower case.
string.split(separator,maxsplit)	Return a list of the words of the string, If the optional second argument sep is absent or None
string.join	Concatenate a list or tuple of words with intervening occurrences of sep.
string.strip()	It returns a copy of the string with both leading and trailing white spaces removed
string.lstrip	Return a copy of the string with leading white spaces removed.
string.rstrip	Return a copy of the string with trailing white spaces removed.
string.swapcase	Converts lower case letters to upper case and vice versa.
string.translate	Translate the characters using table
string.upper	lower case letters converted to upper case.

string.ljust	left-justify in a field of given width.
string.rjust	Right-justify in a field of given width.
string.center()	Center-justify in a field of given width.
string.replace	Return a copy of string s with all occurrences of substring old replaced by new.

Logical operation:-

Python consider empty strings as having Boolean value of 'False' and non-empty strings as 'True'.

And

For 'and' operator if left value is false then left value is returned.
And if left value is true then right value is checked and returned.

OR

For 'or' operator if left value is true then left value is returned.
And if left value is false then right value is checked and returned.
This can be explain by below example.

```
str1 = ''
str2 = 'geeks'
print(repr(str1 and str2))
print(repr(str2 and str1))
print(repr(str1 or str2))
print(repr(str2 or str1))
str1 = 'for'
print(repr(str1 and str2))
print(repr(str2 and str1))
print(repr(str1 or str2))
print(repr(str2 or str1))
```

Output:-

```
''
''
'geeks'
'geeks'
'geeks'
'for'
'for'
'geeks'
```

Some Example on string

```
s = 'hello'
s = "hello"
# multiline strings
s = '''hello'''
s = """hello"""
s = str('hello')
print(s)
# Positive Indexing
s = 'hello world'
```

```
print(s[1])
# Negative Indexing
s = 'hello world'
print(s[-3])
# Slicing
s = 'hello world'
print(s[6:0:-2])
print(s[::-1])  # reversing the strings.
print('delhi' + ' ' + 'mumbai')
print('delhi'*5)
'delhi' != 'delhi'
len('hello world')
max('hello world')    # it will the maximum value based on ASCII value.
min('hello world')    # it will the minimum value based on ASCII value.
sorted('hello world',reverse=True)
s = 'hello world'
print(s.capitalize())
print(s)
s.title()
s.upper()
'Hello Wolrd'.lower()
'HeLlO WorLD'.swapcase()
'my name is nitish'.count('i')
'my name is nitish'.find('x')   # it will return the index number. and
if suppose we r fin that value which not exit in particular string then
it would return "-1"
'my name is nitish'.index('name') # it will return the index number.
same as find the only difference is if suppose we r finding that value
which not exit in particular string then it would return "error".

'my name is nitish'.endswith('sho')
'my name is nitish'.startswith('1my')
name = 'nitish'
gender = 'male'

'Hi my name is {1} and I am a {0}'.format(gender,name)
'nitish1234'.isalnum()
'nitish1234%'.isalnum()
'nitish'.isalpha()
'nitish123'.isalpha()
'123'.isdigit()
'123abc'.isdigit()
'first_name'.isidentifier()
'first-name'.isidentifier()
'hi my name is nitish'.split()
" ".join(['hi', 'my', 'name', 'is', 'nitish'])
'hi my name is nitish'.replace('nitisrgewrhgh','campusx')
'nitish                           '.strip()
```

Output:-

```
hello
e
r
wol
dlrow olleh
delhi mumbai
delhidelhidelhidelhidelhi
False
11
‘w’
‘ ‘
['w', 'r', 'o', 'o', 'l', 'l', 'l', 'h', 'e', 'd', ' ']
Hello world
hello world
Hello World
HELLO WORLD
hello world
3
-1
3
False
False
Hi my name is nitish and I am a male
True
False
True
False
True
False
True
False
['hi', 'my', 'name', 'is', 'nitish']
hi my name is nitish
hi my name is nitish
nitish
```

Set

Set is an unordered collection of data types that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. Python sets are classified into two types Mutable and immutable. A set created with ‘set’ is mutable while the one created with ‘frozen set’ is mutable.

Creating a Set

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a ‘comma’.

Methods	Description
add()	Adds an element to a set
remove()	Removes an element from a set. If the element is not present in the set, raise a KeyError
clear()	Removes all elements form a set
copy()	Returns a shallow copy of a set
pop()	Removes and returns an arbitrary set element. Raise KeyError if the set is empty
update()	Updates a set with the union of itself and others.
union()	Returns the union of sets in a new set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another.

Accessing a Set

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. But we can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Example:-

```
a={'1','2','3'}
b={'2','6',8}
print(a)
print(type(a))
a.add(4)
a.copy()
a.isdisjoint(b)
a.issubset(b)
a.issuperset(b)
a.intersection(b)
print(a)
```

Outputs:-

```
{'1', '3', '2'}
<class 'set'>
<class 'set'>
```

```
{'1', '3', 4, '2'}
False
False
False
{'2'}
```

Function

A function is a block of code that has a name and we can call it instead of writing the whole code. We can create a function and then call it 100 times. This add reusability and modularity to our code.

Types of Function:

1. Buit in function – It is a predefined function such as print(), min() etc.
2. User defined function- It is defined by users that means which we make by our programming codes.
3. Anonymous function- these are called Lambda function because they are not declared with the standard def keyword.

Syntax to declare a function

```
def function_name(argument)
    # function body
   return
```

Here -> def – It is keyword used to declare a function.
function_name – Any name passed to a function.
Arguments- Any value passed to a function.

return- (optinal)- returns value from a function. A python function may or may not return a value. If we want our function to return a value to a function call, we use return statement.
Note- The return statement also denotes that the function has ended. Any code after return is not executed.
Call the function;-
Syntax-   function_name(argument if function has argument).
Examples:-

```
def greet():
  print('Welcome')
greet()

def sum(a,b):
  sum=a+b
  print('sum',sum)
sum(10,2)

def sum(a,b):
  sum=a+b
  return sum
sum(10,2)
```

Output:-

```
Welcome
```

```
sum 12
12
```

Function Arguments

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

Types of function argument:-

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

1. Default argument
2. Keyword arguments (named arguments)
3. Positional arguments
4. Arbitrary arguments (variable-length arguments *args and **kwargs)

1.Default argument- A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

2. Keyword argument- The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

3.Positional argument- We used the Position argument during the function call so that the first argument (or value) is assigned to specific argument and the second argument (or value) is assigned to other specific argument.

4.Arbitrary arguments- In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

1. *args in Python (Non-Keyword Arguments)- It is used to pass a variable number of argument to a function. It is used to pass non-keyworded, variable-length argument list. It allows us to pass more arguments than the number of formal arguments that we previously used.

2. **kwargs in Python (Keyword Arguments)- It is used to pass a keyworded, variable number of argument list. It is used with ** (double star). The reason is that the double star allows us to pass through keyword argument. A keyword argument is where we provide a name to the variable as we pass it into function. It is like dictionary that maps each keyword to the value that we pass alongside it.

Examples:-

```python
#Default Arguments
def func(x,y=10):
  print('\n the value of x is',x)
  print('\n the value of y is',y)
  sum=x+y
  print('sum', sum)
func(12)
```

Output:-

```
the value of x is 12

 the value of y is 10
sum 22
```

```
# Keyword Argument
def person (firstname,lastname):

  print(firstname, lastname)
person('Ravi','Kumar')
```
Output:-

```
Ravi Kumar
```

```
# Position Argument
def person(name,age):
  print('My name is ',name)
  print('My age is ', age)
person('Ravi',27)

def person(name,age):
  print('My name is ',name)
  print('My age is ', age)
person('Ravi', 27)
person('Satish',29)
```
Output:-

```
My name is  Ravi
My age is  27
My name is  Ravi
My age is  27
My name is  Satish
My age is  29
```

```
# Arbitrary argument- *arg
def myfun(name,*arg):
  print(name)
  for i in arg:
    print(i)
myfun('Ravi','satish',27,31)
```
Output: -

```
Ravi
satish
27
31
```

```
# Arbitrary argument- **kwargs
def myfun(name,**kwargs):
  print(name)
  for i,j in kwargs.items():
    print(i)
    print(j)
myfun('Ravi', naam='Satish', age='27', pincode='110031')
```
Output:-

```
Ravi
naam
Satish
```

```
age
27
pincode
110031
```

Lambda function

A lambda function is a anonymous function . A lambda function can take any number of arguments, but can have one expression. It can take any number of arguments.

Syntax:-

Lambda arguments: expression

```
## lambda function
x=lambda a:a+10
print(x(5))

x= lambda a,b:a*b
print(x(5,6))
```

Output:-

```
15
30
```

OOP’s Concept

OOPs stands for Object-oriented programming. OOPs in Python organizes a program around the various objects and well-defined interfaces. The OOPs Concepts in Python are abstraction, encapsulation, inheritance, and polymorphism. These concepts aim to implement real-world entities in programs. OOPs help in creating a working method and variable that can be reused without compromising on security. The emphasis of OOPs concepts is on data rather than on functions and is mainly used in different object-oriented programming languages such as Java, C#, C++, Python, Perl, Ruby, etc.

Advantages of oops

Re-usability, Code maintenance, Data Redundancy, Security, Easy troubleshooting, Problem-Solving, Flexibility and Design Benefits.

Note- Java OOPs Concepts are one of the core development approaches that is widely accepted.

OOPS concept in Python.

1. Class
2. Object
3. Encapsulation
4. Abstraction
5. Inheritance
6. Polymorphism

1. Class

A class is just like a blueprint for objects – it has no values itself. It is an abstract data type. We can have multiple objects of one class.
To define class, we use the class keyword. The class name should be in pascal case.

```
class Car:
  colour='blue'# data
  model='sports'#dat
  def calculate_avg_speed(Km,Time):
    speed=(Km*Time);
    print(speed)
Car.calculate_avg_speed(10,10)# this to call the function/method inside class
```

Output-

```
100
```

Constructor

In OOP’s constructors plays an important role. Constructor is a special method that is used to initialize object. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes. when we create a object inside the constructor then code inside the constructor is automatically called. Benefits of the constructor is that user don't have the access. It will be automatically called or executed when program start. It is used to write configuration related codes.

How to define constructor in Python:

For defining constructor in python we use init by adding double underscore in prefix and postfix like:

def __init__()

Types of Constructors

Constructors in Python can be of two types:

a) Parameterized Constructor:

• Parameterized constructors are ones which have parameters (other than self) defined in

the __init__ method's parameter list. This type of constructor can take arguments from the user.

b) Non-parameterized Constructor in Python:

• It is also known as the default constructor.

• The __init__ method includes a single parameter self. No other parameters are present in __init__'s

parameter list.

• Consequently, this constructor takes no arguments while creating a new object.

Example: -

```
class student:
  def __init__(self):
    self.name='name'
    self.roll='roll'
  def show(self):
    print('Name:',self.name)
    print('Rollno:',self.roll)

st=student()
st.show()
st.name
st.roll
```

Output-

```
Name: name
Rollno: roll
'name'
'roll'
```

Creating an Object in Python

There is no need to use the new keyword for creating objects in Python. Just use this class constructor:

Code:
>>> obj= ClassName()

>>> obj.name
This calls the __init__() method in the ClassName class.
We did not declare it, but it uses the default one.

The __init__() Method

This is a magic method (dunder method) which we can use to initialize values for classes (objects). So __init__() has initialization code. Every class has __init__ and this is executed when we instantiate the class. We can also use this method to do anything you want to do when the object is being created.

```
class student:
  def __init__(self):
    self.name='name'
    self.roll='roll'
  def show(self):
    print('Name:',self.name)
    print('Rollno:',self.roll)

st=student()
st.show()
st.name
st.roll
```

Output-

```
Name: name
Rollno: roll
'name’
‘roll’
```

Python Classes Methods

Python classes can have methods and functions. Functions are simply functions. But methods act on an object and can modify it. Each method has to take the self parameter as the first parameter. It tells it to work on this object. However, you can call it anything you want.

```
class Fruits:
  def __init__(self):
    self.name='fruit'
    self.color='color'
  def show(self):
    print(f'my name is {self.name} and I am {self.color}')

ds=Fruits()
ds.show()
```

Output:-

```
my name is fruit and I am color
```

```
class Fruits:
  def __init__(self,name,color):
    self.name=name
    self.color=color
  def show(self):
    print(f'my name is {self.name} and I am {self.color}')

ds=Fruits('grapes', 'green')
ds.color
ds.show()
```

Output:-

```
my name is grapes and I am green.
```

Reference Variables

- Reference variables hold the objects.
- We can create objects without reference variable as well.
- An object can have multiple reference variables.
- Assigning a new reference variable to an existing object does not create a new object.

```
class Person:

  def __init__(self):
    self.name = 'nitish'
    self.gender = 'male'

Person() #if we write like this  then also our object will be created
p=Person() # if we write like this  then also our object will be
created and we are storing its reference in p. p is variable name which
has memory address of person()/object. p is not object it contain
address of the object . thus it is called as reference varible.so for
storing the object and making it available we are storing it in p.

q=p # q is pointing the object location of p.

# Multiple ref
print(id(p))
print(id(q))
# change attribute value with the help of 2nd object
print(p.name)
print(q.name)
q.name = 'ankit' # here we hve make a changes in q which wi be
bydefault cause change in p . therefore the p.name and q.name both will
be the same 'ankit'
print(q.name)
print(p.name)
```

Output:-

```
<__main__.Person at 0x7e06a32c65f0>
```

```
140655538334992
140655538334992
```

```
nitish
nitish
ankit
ankit
```

Pass by reference

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

Examples:-

```
 class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

# outside the class -> function
def greet(person):  #here we are giving class object as function.
  print('Hi my name is',person.name,'and I am a',person.gender)
  p1 = Person('ankit','male')
  return p1       #And second thing is that fuction is returning an
object.

p = Person('nitish','male')
x = greet(p)
print(x.name)
print(x.gender)
```

Output:-

```
Hi my name is nitish and I am a male
ankit
male
```

```
class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

# outside the class -> function
def greet(person):
  print(id(person))
  person.name = 'ankit'
  print(person.name)

p = Person('nitish','male')
print(id(p))
greet(p)
```

```
print(p.name)
```

Output:-

```
134526280505600
134526280505600
ankit
ankit
```

Encapsulation

Encapsulation in python describes the concept of bundling data and methods with in a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variable) and methods into a single unit. Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding. Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class. Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single underscore and double underscores. • Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

- Public Member: Accessible anywhere from outside of class.
- Private Member: Accessible within the class.
- Protected Member: Accessible within the class and its sub-classes.

Public Member:

```
class Employee:
# constructor
  def __init__(self, name, salary):
# public data members
    self.name = name
    self.salary = salary
# public instance methods
  def show(self):
# accessing public data member
   print("Name: ", self.name, 'Salary:', self.salary)
# creating object of a class
emp = Employee('Jassica', 10000)
# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)
# calling public method of the class
emp.show()
```

Output:

```
Name:  Jassica Salary: 10000
Name:  Jassica Salary: 10000
```

Private Member:-

```
class Employee:
```

```
# constructor
  def __init__(self, name, salary):
# public data member
    self.name = name
# private member
    self.__salary = salary
# creating object of a class
emp = Employee('Jessa', 10000)
# accessing private data members
print('Salary:', emp.__salary)
```

Output:-

```
AttributeError                            Traceback (most recent call
last)
<ipython-input-64-1ee5bbb39515> in <cell line: 11>()
      9 emp = Employee('Jessa', 10000)
     10 # accessing private data members
---> 11 print('Salary:', emp.__salary)

AttributeError: 'Employee' object has no attribute '__salary'
```

Note:- Here we can see we can't access private member.

Managing to access private member

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this _classname__datamember

```
class Employee:
# constructor
  def __init__(self, name, salary):
# public data member
   self.name = name
# private member
   self.__salary = salary
# public instance methods
  def show(self):
# private members are accessible from a class
    print("Name: ", self.name, 'Salary:', self.__salary)
# creating object of a class
emp = Employee('Jessa', 10000)
# calling public method of the class
emp.show()
emp._Employee__salary
```

Output:-

```
Name:  Jessa Salary: 10000
10000
```

Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members. In Python, private variables are not hidden fields like in other programming languages.

The getters and setters methods are often used when:

- When we want to avoid direct access to private variables
- To add validation logic for setting a value.

Examples:-

```
class Student:
  def __init__(self, name, age):
# private member
    self.name = name
    self.__age = age
# getter method
  def get_age(self):
   return self.__age
# setter method
  def set_age(self, age):
    self.__age = age
stud = Student('Jessa', 14)
# retrieving age using getter
print('Name:', stud.name, stud.get_age())
# changing age using setter
stud.set_age(16)
# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

Output:-

```
Name: Jessa 14
Name: Jessa 16
```

Advantages of Encapsulation

- Security: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- Data Hiding: The user would not be knowing what is going on behind the scenes. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- Simplicity: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other. • Aesthetics: Bundling data and methods within a class makes code more readable.

Static methods and Instance method

- Static attributes are created at class level.
- Static attributes are accessed using ClassName.
- Static attributes are object independent. We can access them without creating instance (object) of the class in which they are defined.
- The value stored in static attribute is shared between all instances(objects) of the class in which the static attribute is defined.
- Static Variables remain the same for every object and are defined outside the constructor. . Usually called by   classname .variablename

Instance Variable:-

- Instance variable are somewhat dynamic because they can have different values in each object. They are defined under constructor. Usually called by objectname .variablename

```
class Lion:
  __water_source="well in the circus"

  def __init__(self,name, gender):
      self.__name=name
      self.__gender=gender

  def drinks_water(self):
      print(self.__name,
      "drinks water from the",Lion.__water_source)

  @staticmethod
  def get_water_source():
      return Lion.__water_source

simba=Lion("Simba","Male")
simba.drinks_water()
print( "Water source of lions:",Lion.get_water_source())
```

Output:-

```
Simba drinks water from the well in the circus
Water source of lions: well in the circus
```

Aggregation

It means one class owns the other class. i.e one class will be owner other will be its property. suppose first class is Customer and other class is Address so Customer class owns Address Class i.e Customer class Has A Address class.

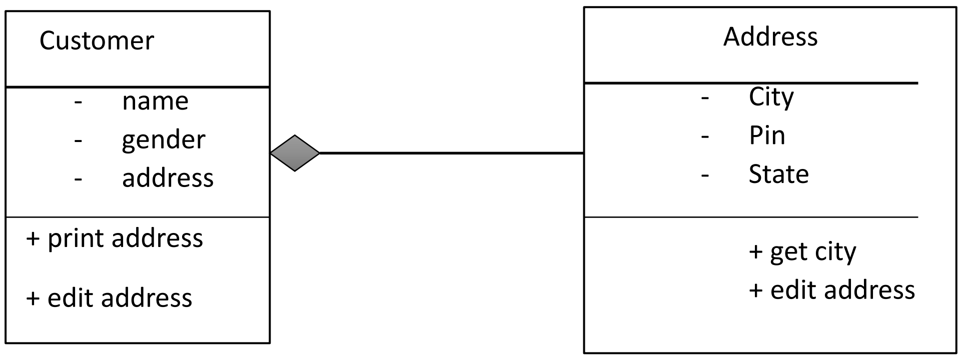
```
class Customer:

  def __init__(self,name,gender,address):
    self.name = name
    self.gender = gender
    self.address = address

  def print_address(self):
    print(self.address.get_city(),self.address.pin,self.address.state)

  def edit_profile(self,new_name,new_city,new_pin,new_state):
    self.name = new_name
    self.address.edit_address(new_city,new_pin,new_state)


class Address:

  def __init__(self,city,pin,state):
      self.__city = city
      self.pin = pin
      self.state = state

  def get_city(self):
    return self.__city

  def edit_address(self,new_city,new_pin,new_state):
    self.__city = new_city
    self.pin = new_pin
    self.state = new_state

add1= Address('hiriyur', '1212', 'karanataka')
cust= Customer('sana','female',add1)
cust.print_address()

print(cust)
print(add1)
cust.print_address()



add1 = Address('gurgaon',122011,'haryana')
cust = Customer('nitish','male',add1)
cust.print_address()
print(cust)

cust.print_address()

cust.edit_profile('ankit','mumbai',111111,'maharastra')
cust.print_address()
```

Output:-

```
hiriyur 1212 karanataka
<__main__.Customer object at 0x7e87c96fa110>
```

```
<__main__.Address object at 0x7e87c96fa170>
hiriyur 1212 karanataka
gurgaon 122011 haryana
<__main__.Customer object at 0x7e87c96f89d0>
gurgaon 122011 haryana
mumbai 111111 maharastra
```

Aggregation class diagram



Inheritance

• In Python, inheritance is the capability of a class to pass some of its properties or methods to

it’s derived class (child class). With inheritance, we build a relationship between classes based on how they are derived.

Type of Inheritance

```
class User:

  def __init__(self):
    self.name = 'nitish'
    self.gender = 'male'

  def login(self):
    print('login')

# child
class Student(User):



  def enroll(self):
    print('enroll into the course')

u = User()
s = Student()

print(s.name)
s.login()
s.enroll()
```

Output-

```
nitish
login
enroll into the course.
```

Types of Inheritance in Python

1.Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

1. Single Inheritance- In single inheritance, a single class inherits from a class. This is the simplest form of inheritance.

```
class Parent:
  def show(self):
    print("Parent method")
class Child(Parent):
  def display(self):
    print("Child method")
c = Child()
c.display()
c.show()
```

output:-

```
Child method
Parent method
```

2. Multilevel Inheritance- Python supports multilevel inheritance, which means that there is no limit on the number of levels that you can inherit. We can achieve multilevel inheritance by inheriting one class from another which then is inherited from another class.

```
class A:
  def show(self):
    print("A method")
class B(A):
  def display(self):
    print("B method")
class c(B):
  print('C method')
c = Child()
c.display()
c.show()
```

Output:-

```
C method
Child method
Parent method
```

3.Multiple Inheritance- Python also allows us to inherit from more than one class. To achieve this we can provide multiple classes separated by commas.

```
class A:
  def show(self):
    print("A method")
class B:
  def display(self):
    print("B method")
class C:
  def method(self):
    print('C method')
class D(A,B,C):
  def method(self):
```

Output:-

```
D method
A method
```

4.Hierarchical Inheritance- In a hierarchical inheritance, a class is inherited by more than one class

```
class A:
  def show(self):
    print("A method")
class B(A):
  def display(self):
    print("B method")
class C(A):
  def method(self):
    print('C method')

c=C()
c.method()
c.show()
```

Output:-

```
C method
A method
```

5.Hybrid Inheritance- The term Hybrid describes that it is a mixture of more than one type. Hybrid inheritance is a combination of different types of inheritance.

```
class A:
  def show(self):
    print("A method")
class B(A):
  def display(self):
    print("B method")
class C(A):
  def method(self):
    print('C method')
class D(B,C):
  def method(self):
    print('D method')


c=D()
c.show()
c.method()
```

Output: -

```
A method
D method
```

Super () function

Method overriding is an important concept in object-oriented programming. Method overriding allows us to redefine a method by overriding it.

• For method overriding, we must satisfy two conditions:

• There should be a parent-child relationship between the classes.

• Inheritance is a must.

• The name of the method and the parameters should be the same in the base and derived class in order to override it.

```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    def buy(self):
        print ("Buying a smartphone")

s=SmartPhone(20000, "Apple", 13)
p=Phone(3000,'Ess',21)
p.buy()
s.buy()
```

Output: -

```
Inside phone constructor
Inside phone constructor
Buying a phone
Buying a smartphone
```

Note: - Overloading Methods in Python - Python doesn’t support method overloading.