Python ClassNotes

Monday, March 7, 2022 10:33 AM

- Python is case sensitive language
- # is used for commenting
- Tuple is immutable and list is mutable
- Python have some datatypes like list string tuple dictionary set.
- Dictionary is similar to mapping in python
- In python indentation is important
- If we want to pass multiple parameters into same argument (to store variable number of arguments)

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print (arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

- Shallow copy vs deep copy vs =
- Python is interpreter language
- Object oriented
- Open source
- Simple to understand
- Java is faster than python
- Large supportive community
- Data type is chosen at runtime either dynamically or manually.
- Case sensitive language.
- In python there is no necessity of brackets , the syntax is managed through indentation.
- Set stores only unique values.

```
# Python program to illustrate
# *args with first extra argument
def myFun(arg1, *argv):
    print ("First argument :", arg1)
    for arg in argv:
        print("Next argument through *argv :", arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

- Or we can accept some argument's through variable declaration and rest through argv
- To take input from the user

```
val = input("Enter your value: ")
print(val)
```

Some main coding concepts of python

```
list vs tupr:
>>> mylist=[1,3,3]
>>> mylist[1]=2
>>> mytuple=(1,3,3)
>>> mytuple[1]=2
Eroor in mytuple[1]=2
TypeError: 'tuple' object does not support item assignment

Unlike C++, we don't have ?: ternary operator in py
Instead we have [on true] if [expression] else [on false]
>>> a,b=2,3
>>> min=a if a<b else b
>>> min

Negative index: unlike +vs starts searching from right
ex: mylist[-3]
or for slicing
mylist[-6:-1] or (1,2,3,4,5)[-2:-4]

print the last 4 numbers from the array below:
arr = np.array([1,2,3,4,5,6,7]) for this use: print(arr[3:])

Identifier:
beginning char: only _ or alphabet
rest char: anything
keywords: cannot be used as identifiers

'AyuShi'.lower() .isupper()
'AyuShi'.swapcase()

pass stmt:
There may be times in our code when we haven't decided what to do yet, but we must type something for it to be syntactically correct.


>>> import copy
>>> help(copy.copy)
print(dir(copy))
deep copy vs shallow copy
in shallow copy any changes made to a copy of object do reflect in the original object.

dir() function displays all the members of an object(any kind).

>>> mydict={'a':1,'b':2,'c':3,'e':5}
>>> mydict.keys()
>>> mydict.values()
or
>>> roots={x**2☺ for x in range(5,0,-1)}
>>> roots
```

Copy Operator vs equality operator in python

```
numbers = [1, 2, 3, 4, 5]

new_numbers = numbers

new_numbers[0] = 9

print('Numbers list:', numbers)
print('ID of numbers:', id(numbers))

print('New numbers:', new_numbers)
print('ID of new numbers:', id(new_numbers))
```

- Here we can see that, we created a list called numbers and a new list called new_numbers and assigned new_numbers with the numbers but when we change anyone of those lists then we can see both the lists are affected hence here we can observe that using equality operator we are copying the address of the list not the copy of it. To prevent this we use copy

```
import copy

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

new_list = copy.copy(old_list)

new_list[0] = ['a','b','c']
```

```python
import copy

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

new_list = copy.copy(old_list)

new_list[0] = ['a','b','c']

print("Old list:", old_list)
print("New list:", new_list)
```

- The following is copying using copy module. Here shallow copy creates a copy of the object but references each element of the object.
- In above code output we get two lists in first list first element is 1,2,3 and in second list first element is a , b , c .
- But when we try to change the inner element's then both the lists change because shallow copy creates a copy of the object but references each element of the object.

```python
import copy

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

new_list = copy.copy(old_list)

new_list[0][2] = 'c'

print("Old list:", old_list)
print("New list:", new_list)
```

```
Python - copying.py:7 ✓

Old list: [[1, 2, 'c'], [4, 5, 6], [7, 8, 9]]
New list: [[1, 2, 'c'], [4, 5, 6], [7, 8, 9]]
[Finished in 0.83s]
```

- So to prevent it we use deep copy.
- Deep copy creates a copy of the object and the elements of the object. But shallow copy

doesn't create copy of elements of the object.

- Following is the example of deep copy

```
import copy

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

new_list = copy.deepcopy(old_list)

new_list[0][2] = 'c'

print("Old list:", old_list)
print("New list:", new_list)
```

```
Python - copying.py:5 ✔

Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list: [[1, 2, 'c'], [4, 5, 6], [7, 8, 9]]
[Finished in 0.899s]
```

- Some more operations in python

```
>>> roots={x**2😉 for x in range(5,0,-1)}
>>> roots
{25:5,16:4,9:3,4:2,1:1}

check if all chars in string are alpha-numerals: isalnum()
capitalise 1st letter only: .capitalize()
'123.3'.isdigit() - returns a non-zero value for '0' to '9' and zero for others
'123'.isnumeric() - True if all characters in a string are numeric characters
'Ayushi'.istitle() - True if all words in a text start with a upper case letter
' '.isspace()
'123F'.isdecimal()
7//2 - returns integer part of division(floor division) => 3
a.insert(2,3) - insert 3 at position 2
a.reverse() or >>> a[::-1]  >>> a
max('flyiNg') - to get max alphabetical char frm a string - based on ascii values
complex(3.5,4) - o/p 3.5+4j
eval('print(max(22,22.0)-min(2,3))') - similar to eval in intellij - parse string as an expression
hash(3.7) - has is similar to fingerprint, returns unique value for wach obj ex : define a hash value length of
hex(14)
input('Enter a number')
len('Ayushi') - returns length of obj(can be any obj)
file=open('tabs.txt') - opens a file
>>> word='abcdefghij'
>>> word[:3]+word[3:] - o/p is 'abcdefghij' frst slice gives abc the next gives defghij
To convert a list into a string
>>> nums=['one','two','three','four','five','six','seven']
>>> s=' '.join(nums)
>>> s
o/p 'one two three four fi
To remove duplicate elem
>>> list=[1,2,1,3,4,2]
>>> set(list)
```

it will get what is the value of this particular expression and

```
>>> list=[1,2,1,3,4,2]
>>> set(list)

bytes([2,4,8]) => returns an immutable bytes =>o/p is b'\x02\x04\x08'

list(zip(['a','b','c'],[1,2,3])) => [('a', 1), ('b', 2), ('c', 3)]
flush method in python:
The flush() method in Python file handling clears the internal buffer of the file.
In Python, files are automatically flushed while closing them.
However, a programmer can flush a file before closing it by using the flush() method.
print('09','12','2016', end='', flush=True ,sep='-') => 09-12-2016
end is used to enter somthing at the end of the line and this prints the input in a single line
sep acts as a separator between the comma separated values in print stmt
end='\n' #\n provides new line after printing the year


'hi'<'Hi' - returns True if left value is lesser
>>> a=7
>>> a+=1 => a=8

int('227') or type(int('227')) - convert string to int
type('227') - to check type

int(x)
bin(x)
oct(x)
hex(x)
list.append(x)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(-5))
[]
```

```
datatypes:
num,strng
list,tuple
dict,file


docstring: just for doc purpose(place it as the first thing under a function)
ex:
"""
The function prints Hi
"""
To get a function's docstring,
sayhi.__doc__
```

```
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinIm
similarly for globals\
ex:
age = 23
globals()['age'] = 25
print('The age is:', age)
Output
The age is: 25


limitations:
dynamically-typed
weak in mobile computing
interpreted nature imposes a speed penalty
underdeveloped database access layers - not a good choice for db based appns

appns:
Web and Internet Development
Desktop GUI
Scientific and Numeric Applications
Software Development Applications
Applications in Education
Applications in Business
Database Access
Network Programming
Games, 3D Graphics
Other Python Applications

>>> import os
>>> os.getcwd() - get current directory(cwd - current working directory)
>>> os.chdir('C:\\Users\\lifei\\Desktop')
```

```
>>> import os
>>> os.getcwd() - get current directory(cwd - current working directory)
>>> os.chdir('C:\\Users\\lifei\\Desktop')

Python interpreter prompt - >>> - If you have worked with the IDLE
Python does not support curly braces

def add(a,b):
   return a+b

do-while loop:
Python doesn't have do-while loop. But we can create a program like this.
The do while loop is used to check condition after executing the statement.
It is like while loop but it is executed at least once.

lambda:
lambda is a function
it can take any no of arguments bt can only have one expression.
x = lambda a : a + 10    => similar to return a+10 , when input param is a
print(x(5))


recursion:
>>> def facto(n):
if n==1: return 1
return n*facto(n-1)
>>> facto(4)
```

```
[i for i in range(1,11,2)] => [1, 3, 5, 7, 9] # range of menas a range of values
s= 'I love python'
for i in s:
 if i!=' ': print(i,end='') => Ilovepython
for i in range(6):
 print(s)

PEP 8 is a coding convention that lets us write more readable code.

>>> a,b=2,3
>>> a,b=b,a
>>> a,b

>>> a,b,c=3,4,5 #This assigns 3, 4, and 5 to a, b, and c respectively
>>> a=b=c=3 #This assigns 3 to a, b, and c

Python files first compile to bytecode. Then, the host executes them.

Technical Python Interview Questions and Answers - to check interview qns
```

```
#%% - cell break - ctrl enter runs the cell only
ex :
#%%
#importing codes
from math import *
print(floor(3.7))
print(ceil(3.7))
print(sqrt(4))


import math

difference between import and from import in python:
import imports the whole library. from import imports a specific member or members of the library.
from datetime import date
import datetime


Ctrl+C - to break infinite loop in py

pass by refernce for py
 However, when we pass literal arguments like strings, numbers, or tuples, they pass by value
the change reflects in the calling function
#!/usr/bin/python
# Function definition is here
def changeme( mylist ):
"This changes a passed list into this function"
mylist.append([1,2,3,4]);
print "Values inside the function: ", mylist
return
# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
```

```
pass by ref:
Values inside the function: [10,20,30,1,2,3,4]
Values outside the function: [10,20,30,1,2,3,4]
pass by value:
Values inside the function: [10,20,30,1,2,3,4]
Values outside the function: [10,20,30]


with statement
ensures that cleanup code is executed when working with unmanaged resources by encapsulating common preparation a


with open('data.txt') as data:


OOPS:
Encapsulation
Abstraction
Inheritance
Polymorphism
Data hiding

•Numpy adds to Python support for large, multi-dimensional arrays and matrices, along with a large library of hi
•SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Pyth
•Pandas is a library written for data manipulation and analysis in Python. Offers data structures and operations
•Scikit-learn is a Python module for machine learning built on top of SciPy
Numpy - support for large, multi-dimensional arrays and matrices - high-level MATHEMATICAL functions
SciPy - mathematical ALGORITHMS and convenience functions - built on the Numpy
Pandas - data manipulation and analysis - DATA STRUCTURES and operations for manipulating numerical tables and t
Scikit - module for machine learning - built on top of SciPy
```

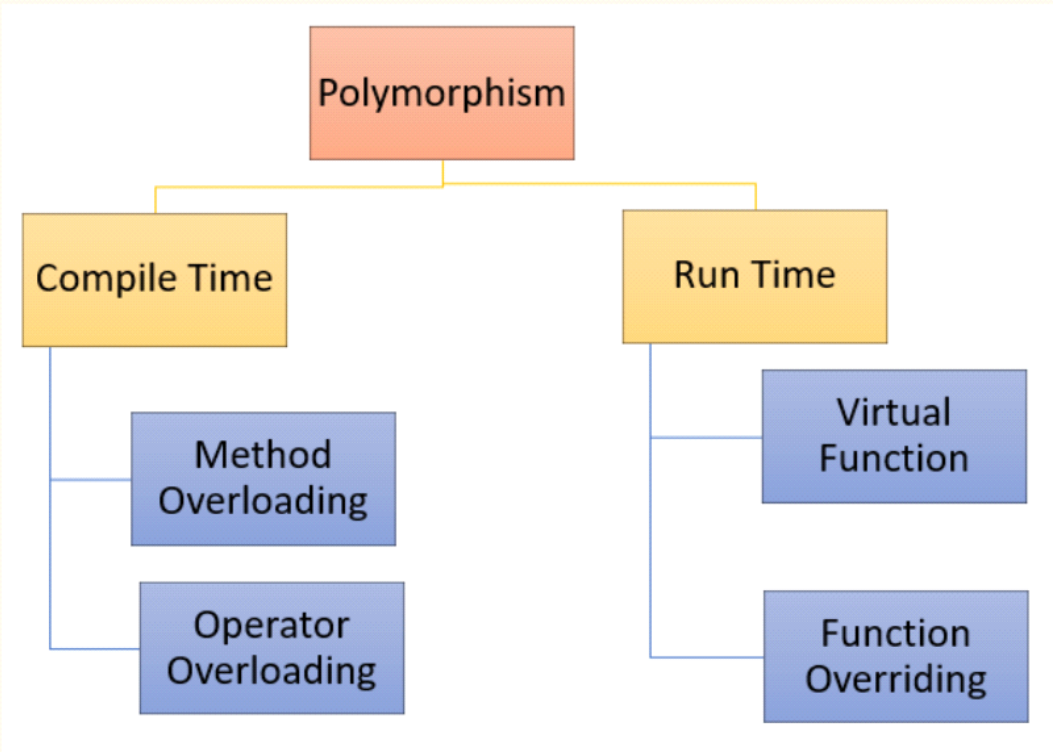
```
Tasks can be:
created
edited
ended (when the task is running)
deleted
disabled and enabled back
exported and imported (in .xml file)



Three necessary ingredients best kept separated inside their own function definitions:
program logic (your script turned to function[s])
ui definition (argparse.ArgumentParser creation)
putting things together (main() function that takes parsed arguments and feeds them into relevant function)


Remember to finish your script with the conditional statement:
if __name__ == "__main__":
    main()
This way python interpreter will execute main() function when the script gets executed from the command line


Numpy


import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr[4:]) # Slice elements from index 4 to the end of the array:
print(arr[4:]) # Slice elements from index 4 to the end of the array:
print(arr[:4]) # Slice elements from the beginning to index 4
print(arr[-3:-1]) # Slice from the index 3 from the end to index 1 from the end:
arr = np.array([1, 2, 3, 4], dtype='S') # to create array with datatype string
```

Polymorphism causes a member function to behave differently based on the object that calls or invokes it. It also mean a function name can have many forms. It occurs when you have a hierarchy of classes related through inheritance.

```
polymorphism

class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro() //There are many types of birds.
obj_bird.flight()  //Most of the birds can fly but some cannot.

obj_spr.intro() //There are many types of birds.
obj_spr.flight()  //Sparrows can fly.

obj_ost.intro()  //There are many types of birds.
obj_ost.flight()  //Ostriches cannot fly.
```

* Here flight is the function which exhibits multiple behaviors when the calls are received from different type of objects. If we call flight through ostrich we get to print Ostriches cannot fly and if through sparrow it will be sparrows can fly and if intro called through sparrow or ostrich the parent class method executes and if we call flight function through parent class object it print Most of the birds can fly but some cannot. Here the flight function have many forms hence it exhibits the polymorphism. Python exhibits only one type of polymorphism that is called run time polymorphism.

- In general we have two types of polymorphism they are compile time polymorphism and run time polymorphism.



Compile Time polymorphism:
You invoke the overloaded functions by matching the number and type of arguments. The information is present during compile-time. compiler will select the right function at compile time. Compile-time polymorphism is achieved through function overloading and operator overloading.

- It is used for method overloading, it support compile time binding and python does not support compile time polymorphism. In general, class , method and object are bind at compile time.

Why python doesn't support method overloading?

- Python doesn't support method overloading , it is not possible to declare more than one method with same number of parameters with same function name. Because method arguments in python doesn't have any type. In python as the variable doesn't have any datatype it may be anything like integer , string , float or double.
- If we try to achieve method overloading with different number of parameters then we will have error because python only remembers the last function that is declared for each function name. So now add function with three parameters remembered but add function with two parameters will be forgotten. So if we try to execute it, we get an error.

```
class math:

    def add(self,a,b):

        c=a+b

        print("Addition of two Number:",c)

    def add(self,a,b,c):

        d=a+b+c

        print("Addition of three Number:",d)

#creating the object of the class

m=math()

#m.add(2,3)

m.add(5,4,6)
```

To achieve Method overloading in python:-

There are different ways to achieve method overloading in python.But we will discuss one of them.

1. By using normal ways**(Not more efficient)**
2. By using Variable arguments **(efficient)**
3. By using Multiple Dispatch Decorator **(efficient)**

1.) By using Variable arguments:-By using variable arguments,we can achieve **method overloading** concepts in python.

e.g.

```
class math:
    def add(self,*n):
        sum=0
        for num in n:
            sum=sum+num
        return sum
#creating the object of the class
m=math()
#method with four argument
sum1=m.add(3,4,5,6)
print("sum of four arguments="+str(sum1))
sum2=m.add(10,20,30)
print("sum of three arguments="+str(sum2))
sum3=m.add(40,50)
print("sum of two arguments="+str(sum3))
```

2.) Run-time polymorphism:-

- It involves deciding at runtime which function from base class should get called.
- Python supports runtime polymorphism. Python supports method overriding and operator overloading.

Three ways to implement polymorphism concepts in python

1) Polymorphism with function and objects

3.) Polymorphism with Inheritance:-

Polymorphism defines methods in derived class that have the same name as the method in the parent class.In Inheritance,we know that the child class inherits the methods from the parent or base class.It is also possible to modify a method in a derived class that has inherited from the parent class.We already know,some times ,the method inherited from the parent or base class doesn't fit the derived class.

This process to modify a method in the child class is known as method overriding.

Inheritance:

```
inheritance



# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)
```

```
    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Encapsulation

```
encapsulation

# Python program to
# demonstrate private members

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a) //GeeksforGeeks
obj2 = Derived() //attr err
print(obj1.c) //attr err

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
```

Assignment 1:

```
1. Regex in Python to put spaces between words starting with capital letters soln:

import re

def putSpace(input):

    # regex [A-Z][a-z]* means any string starting
    # with capital character followed by many
    # lowercase letters
    words = re.findall('[A-Z][a-z]*', input)//BruceWayneIsBatman

    # Change first letter of each word into lower
    # case
    for i in range(0,len(words)):
      words[i]=words[i][0].lower()+words[i][1:]//bruce
      print(' '.join(words))// bruce wayne


# Driver program
if __name__ == "__main__":
    input = 'BruceWayneIsBatman'
    putSpace(input)


    //bruce wayne is batman
```

```
 2. Extract IP address from file using Python https://www.geeksfo

# importing the module
import re

# opening and reading the file
with open('C:/Users/user/Desktop/New Text Document.txt') as fh:
   fstring = fh.readlines()

# declaring the regex pattern for IP addresses
pattern = re.compile(r'(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})')

# initializing the list object
lst=[]

# extracting the IP addresses
for line in fstring:
   lst.append(pattern.search(line)[0])

# displaying the extracted IP addresses
print(lst)



123.10.10.10    19.19.19.19
12.12.12.12
```

Quick Sort algorithm: (Quick sort in 4 minutes**)**

Selection Sort Algorithm: (Selection Sort | GeeksforGeeks**)**

Inheritance

Single Inheritance : If class B extends Class A then it is single level inheritance

```
# Python program to demonstrate
# single inheritance


# Base class
class Parent:
     def func1(self):
          print("This function is in parent class.")

# Derived class
class Child(Parent):
     def func2(self):
          print("This function is in child class.")

# Driver's code
object = Child()
object.func1()
object.func2()

o/parent
This function is in parent class.
This function is in child class.
```

Multi level Inheritance: If class C extends class B and class B extends class A

```
multilevel

# Python program to demonstrate
# multilevel inheritance

# Base class
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class
class Son(Father):
    def __init__(self,sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)

        print("Father name :", self.fathername)
        print("Son name :", self.sonname)

#  Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Hierarchical Inheritance: If two or more classes extends same class then it is hierarchical inheritance. If class B and class C extends the class A.

```
hierachial

# Python program to demonstrate
# Hierarchical inheritance


# Base class
class Parent:
      def func1(self):
          print("This function is in parent class.")

# Derived class1
class Child1(Parent):
      def func2(self):
          print("This function is in child 1.")

# Derivied class2
class Child2(Parent):

      def func3(self):
          print("This function is in child 2.")

# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
Output:

This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Multiple Inheritance :If a class extends two or more classes

```
multiple:

# Python program to demonstrate
# multiple inheritance


# Base class1
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Hybrid Inheritance (multiple inheritance + hierarchical inheritance)

```
# Python program to demonstrate
# hybrid inheritance


class School:
     def func1(self):
         print("This function is in school.")

class Student1(School):
     def func2(self):
         print("This function is in student 1. ")

class Student2(School):
     def func3(self):
         print("This function is in student 2.")

class Student3(Student1, School):
     def func4(self):
         print("This function is in student 3.")

# Driver's code
object = Student3()
object.func1()
object.func2()


Output:

This function is in school.
This function is in student 1.
```

Python Variables and Expressions - Python SL

Tuesday, March 8, 2022 4:06 PM

- Full featured object oriented language
- Easy to create custom data objects
- Python have data structures
- Python has several built-in data structures, including lists, dictionaries, and sets, that we use to build customized objects. In addition, there are a number of internal libraries, such as collections and the math object, which allow us to create more advanced structures as well as perform calculations on those structures
- Interactive console
- C/C++ or Java, where the write-compile-test-recompile cycle can increase development time considerably compared to Python's read - evaluate - print loop. Being able to type in expressions and get an immediate response can greatly speed up data science tasks
- Variables are labels attached to objects; they are not the object itself. They are not containers for objects either. A variable does not contain the object, rather it acts as a pointer or reference to an object

```
In [1]: a=[2,4,6]

In [2]: b=a

In [3]: a.append(8)

In [4]: b
Out[4]: [2, 4, 6, 8]
```

- Here we have created a variable, a, which points to a list object. We create another variable, b, which points to this same list object. When we append an element to this list object, this change is reflected in both a and b.
- when we initialize variables in Python, we do not need to declare a type

```
In [1]: a=1

In [2]: type(a)
Out[2]: int

In [3]: a=a+0.1

In [4]: type(a)
Out[4]: float
```

Variable Scope

- It is important to understand the scoping rules of variables inside functions. Each time a function executes, a new local namespace is created. This represents a local environment that contains the names of the parameters and variables that are assigned by the function. To resolve a namespace when a function is called, the Python interpreter first searches the local namespace (that is, the function itself) and if no match is found, it searches the global namespace.
- This global namespace is the module in which the function was defined. If the name is still not found, it searches the built-in namespace. Finally, if this fails then the interpreter raises a NameError exception

```
a=10; b=20
def my_function():
    global a
    a=11; b=21
my_function()
print(a) #prints 11
print(b) #prints 20
```

- We need to tell the interpreter, using the keyword global, that inside the function, we are referring to a global variable. When we change this variable to 11, these changes are reflected in the global scope. However, the variable b we set to 21 is local to the function, and any changes made to it inside the function are not reflected in the global scope. When we run the function and print b, we see that it retains its global value.

Flow control and iteration

Tuesday, March 8, 2022 4:37 PM

- There are two main ways of controlling the flow of program execution, conditional statements and loops.
- The if, else, and elif statements control the conditional execution of statements. The general format is a series of if and elif statements followed by a final else statement

```
x='one'
if x==0:
    print('False')
elif x==1:
    print('True')
else: print('Something else')
#prints 'Something else'
```

- We can also control the flow of execution using loops

```
In [5]: x=0

In [6]: while x < 3 : print(x); x +=1
0
1
2
```

Higher Order Functions

Tuesday, March 8, 2022 4:45 PM

- Python 3 contains two built-in higher order functions, filter() and map()

Note the use of the `lambda` anonymous function:

```
In [40]: lst=[1,2,3,4]

In [41]: list(map(lambda x: x**3, lst))
Out[41]: [1, 8, 27, 64]
```

Similarly, we can use the `filter` built-in function to filter items in a list:

```
In [43]: list(filter((lambda x: x<3),lst))
Out[43]: [1, 2]
```

```
In [19]: words=str.split('The longest word in this sentence')

In [20]: sorted(words, key=len)
Out[20]: ['in', 'The', 'word', 'this', 'longest', 'sentence']
```

Here is another example for case-insensitive sorting:

```
In [84]: sl=['A','b','a', 'C', 'c']

In [85]: sl.sort(key=str.lower)

In [86]: sl
Out[86]: ['A', 'a', 'b', 'C', 'c']

In [87]: sl.sort()

In [88]: sl
Out[88]: ['A', 'C', 'a', 'b', 'c']
```

- Note the difference between the list.sort() method and the sorted built-in function. list.sort(), a method of the list object, sorts the existing instance of a list without copying it. This method changes the target object and returns None.
- On the other hand, the sorted built-in function returns a new list. It actually accepts any iterable object as an argument, but it will always return a list. Both list sort and sorted take two optional keyword arguments as key.

```
In [92]: items.sort(key=lambda item: item[1])

In [93]: items
Out[93]: [['flour', 1.9, 5], ['rice', 2.4, 8], ['Corn', 4.7, 6]]
```

Here we have sorted the items by price.

- Generator expressions, however, do not create a list, they create a generator object. This object does not create the data, but rather creates that data on demand. This means that generator objects do not support sequence methods such as append() and insert(). You can, however, change a generator into a list using the list() function:

```
# compares the running time of a list compared to a generator
import time
#generator function creates an iterator of odd numbers between n and m
def oddGen(n, m):
    while n < m:
        yield n
        n += 2
#builds a list of odd numbers between n and m
def oddLst(n,m):
    lst=[]
    while n<m:
        lst.append(n)
        n +=2
    return lst
#the time it takes to perform sum on an iterator
t1=time.time()
sum(oddGen(1,1000000))
print("Time to sum an iterator: %f" % (time.time() - t1))

#the time it takes to build and sum a list
t1=time.time()
sum(oddLst(1,1000000))
print("Time to build and sum a list: %f" % (time.time() - t1))
```

This prints out the following:

```
Time to sum an iterator: 0.133119
Time to build and sum a list: 0.191172
```

Typically, generator objects are used in `for` loops. For example, we can make use of the `oddcount` generator function created in the preceding code to print out odd integers between `1` and `10`:

```
    for i in oddcount(1,10):print(i)
```

```
In [5]: lst1= [1,2,3,4]

In [6]: gen1 = (10**i for i in lst1)

In [7]: gen1
Out[7]: <generator object <genexpr> at 0x000001B981504C50>

In [8]: for x in gen1: print(x)
10
100
1000
10000
```

Classes and object programming

Tuesday, March 8, 2022 5:29 PM

- Classes are a way to create new kinds of objects and they are central to object-oriented programming. A class defines a set of attributes that are shared across instances of that class. Typically, classes are sets of functions, variables, and properties.
- By organizing our programs around objects and data rather than actions and logic, we have a robust and flexible way to build complex applications.
- It is important to understand that defining a class does not, by itself, create any instances of that class. To create an instance, a variable must be assigned to a class
- The functions defined inside a class are called instance methods. They apply some operations to the class instance by passing an instance of that class as the first argument. This argument is called self by convention, but it can be any legal identifier. Here is a simple example:
- Class variables, such as numEmployee, share values among all the instances of the class. In this example, numEmployee is used to count the number of employee instances

```
class Employee(object):
    numEmployee = 0
    def __init__(self, name, rate):
        self.owed = 0
        self.name = name
        self.rate=rate
        Employee.numEmployee += 1

    def __del__(self):
        Employee.numEmployee -= 1

    def hours(self, numHours):
        self.owed += numHours * self.rate
        return("%.2f hours worked" % numHours)

    def pay(self):
        self.owed = 0
        return("payed %s " % self.name)
```

- We can create instances of the Employee objects, run methods, and return class and instance variables by doing the following:

```
In [3]: emp1=Employee("Jill", 18.50)

In [4]: emp2=Employee("Jack", 15.50)

In [5]: Employee.numEmployee
Out[5]: 2

In [6]: emp1.hours(20)
Out[6]: '20.00 hours worked'

In [7]: emp1.owed
Out[7]: 370.0

In [8]: emp1.pay()
Out[8]: 'payed Jill '
```

- The methods that begin and end with two underscores are called special methods.
- when we use the + operator, we are actually invoking a call to __add__(). For example, rather than using my_object.__len__() we can use len(my_object) using len() on a string object is actually much faster because it returns the value representing the object's size in memory,

rather than making a call to the object's __len__ method

```
class my_class():
    def __init__(self, greet):
        self.greet = greet
    def __repr__(self):
        return 'a custom object (%r)' % (self.greet)
```

When we create an instance of this object and inspect it, we can see we get our customized string representation. Notice the use of the `%r` format placeholder to return the standard representation of the object. This is useful and best practice, because, in this case, it shows us that the `greet` object is a string indicated by the quotation marks:

```
In [13]: a=my_class('giday')

In [14]: a
Out[14]: a custom object ('giday')
```

Inheritance

Tuesday, March 8, 2022 6:20 PM

- It is possible to create a new class that modifies the behavior of an existing class through inheritance. This is done by passing the inherited class as an argument in the class definition. It is often used to modify the behavior of existing methods, for example:

```
class specialEmployee(Employee):
    def __init__(self,name,rate, bonus):
        Employee.__init__(self, name, rate) #calls the base classes
        self.bonus = bonus

    def hours(self, numHours):
        self.owed += numHours * self.rate + self.bonus
        return("%.2f hours worked" % numHours)
```

- Notice that the methods of the base class are not automatically invoked and it is necessary for the derived class to call them. We can test for class membership using the built-in function isinstance(obj1, obj2). This returns true if obj1 belongs to the class of obj2 or any class derived from obj2.
- There are some methods that are declared in the classes that are not operating on instances. They are static and class methods.
- A static method is just an ordinary function that just happens to be defined in a class. It does not perform any operations on the instance and it is defined using the @staticmethod class decorator. Static methods cannot access the attributes of an instance
- Class methods operate on the class itself, not the instance, in the same way that class variables are associated with the classes rather than instances of that class. They are defined using the @classmethod decorator, and are distinguished from instance methods in that the class is passed as the first argument. This is named cls by convention.
- First parameter of Class method is cls and First parameter of normal methods or instance methods is self keyword.

Static method

- Static methods are simple functions with no self argument.
- Nested inside the class. It only work on the class attributes but not instance attributes.
- It can be called through both class and instance.
- Built in functin staticmethod() is used to create them.

```
class foo(object):
    x =1
    u =1

    @staticmethod
    def average(*mixes):
        return sum(mixes)/len(mixes)

    @staticmethod
    def static_method():
        return foo.average(foo.x,foo.u)

    @classmethod
    def class_method(cls):
        return cls.average(cls.x,cls.u)

a = foo()
print(a.static_method())
print(a.class_method())
```

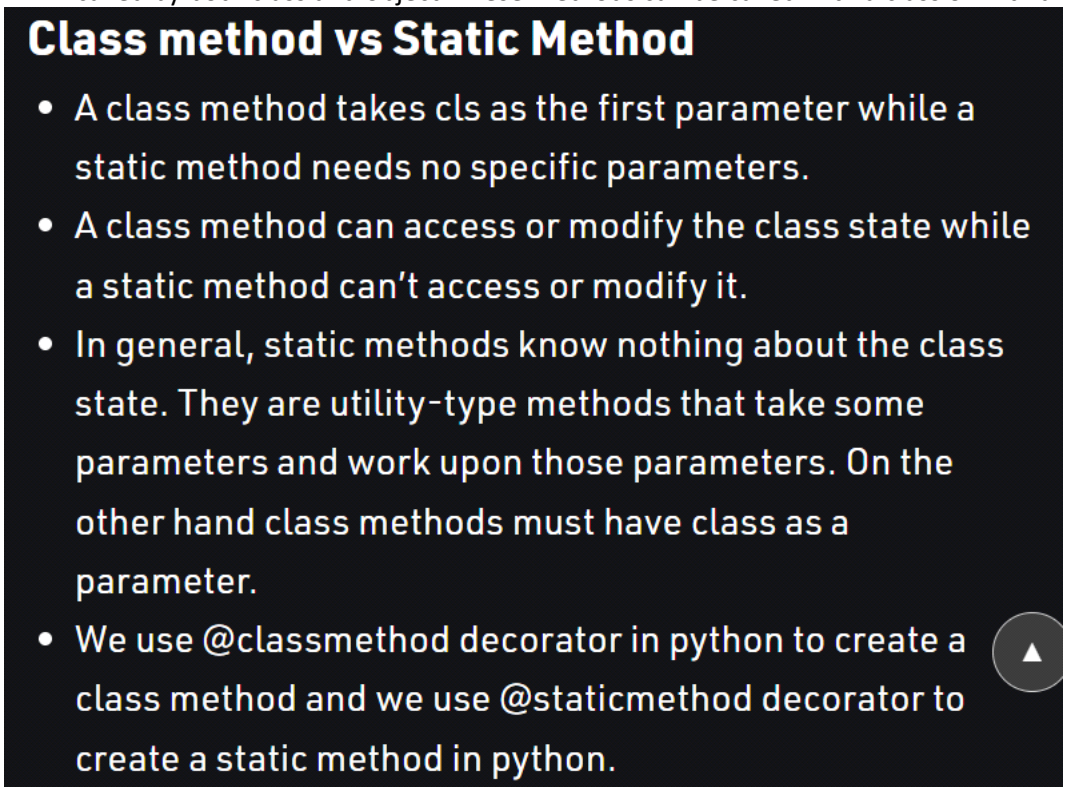
Class Method

Syntax: *classmethod(function)*

Parameter : *This function accepts the function name as a parameter.*

Return Type: *This function returns the converted class method.*

- classmethod() methods are bound to a class rather than an object. Class methods can be called by both class and object. These methods can be called with a class or with an object.



Example of classmethod in Python

Example 1: Create a simple classmethod

In this example, we are going to see a how to create classmethod, for this we created a class with geeks name with member variable course and created a function purchase which prints the object.

Now we passed the method geeks.purchase into classmethod which converts the methods to a class method and then we call the class function purchase without creating a function object.

Python3

```
class geeks:
    course = 'DSA'

    def purchase(obj):
        print("Puchase course : ", obj.course)


geeks.purchase = classmethod(geeks.purchase)
geeks.purchase()
```

Output:

```
Puchase course :  DSA
```