OOP concepts

11.1 document string

Python documentation strings make it easy to link documentation to modules, functions, classes, and methods in Python.
There are two types of document strings.

- one-line docstrings
- multi-line docstrings

One-line Docstrings

One-line docstrings are used to give single-line documentation.one-line docstrings are represented by triple double-quotes.

Example:

```
def sum(a, b):

        """Returns sum of a and b."""

        return a+b

print(sum.__doc__)
```

output:

```
Returns sum of a and b.
```

Multi-line Docstrings

If we want to give more summary and details about function, method, or class we use multi-line docstrings.multi-line docstrings are represented by triple single quotes.

Example:

```
def sum(a, b):

        ‘’’ function name: sum

        Parameters:2
```

```
            Returning sum of two numbers’’’
            return a+b
print(sum.__doc__)
```

output:

```
function name: sum
Parameters:2
Returning sum of two numbers
```

11.2 Inheritance and Types

Inheritance is the capability of one class to derive or inherit the properties from another class.

There are different types of inheritance

- Single inheritance
- Multiple-level inheritance
- Multiple inheritance

Single inheritance

When a child class inherits from only one parent class, it is called single inheritance.

Example:

```
class A:
            def __init__(self, n ):
                        self.n = n
            def display(self):
                        print(self.n*2)
class B(A):
            def __init__(self, roll):
                self.roll = roll
                A.__init__(self,roll)
object = B(23)
object.display()
```

output:

```
46
```

Here we are calling the parent constructor in the child constructor. If you forget to invoke the __init__() of the parent class then its instance variables would not be available to the child class.

Multi-level inheritance

when we have more than two levels of inheritance then we call it multi-level inheritance.

Example:

```
class A(object):
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
class B(A):
    def __init__(self, name, age):
        A.__init__(self, name)
        self.age = age
    def getAge(self):
        return self.age
```

```
class C(B):
    def __init__(self, name, age, address):
        B.__init__(self, name, age)
        self.address = address
    def getAddress(self):
        return self.address
g = C("name", 23, "banglore")
print(g.getName(), g.getAge(), g.getAddress())
```

output:

```
name 23 banglore
```

Here class b is inheriting class a and class c is inheriting class b.

Multiple inheritance

When one child class inherits from multiple parent classes then we call it as Mutiple inheritance.

Example:

```
class A(object):
    def __init__(self):
        self.st1 = "A"
        print("1st class")

class B(object):
    def __init__(self):
```

```
                self.st2= "B"

                print("2nd class")

class C(A, B):

        def __init__(self):

                A .__init__(self)

                B.__init__(self)

                print("Derived class")


        def printStrs(self):

                print(self.st1, self.st2)

ob = C()

ob.printStrs()
```

output:

```
1st class

2nd class

Derived class

A B
```

11.3 Data Hiding(Access Specifier)

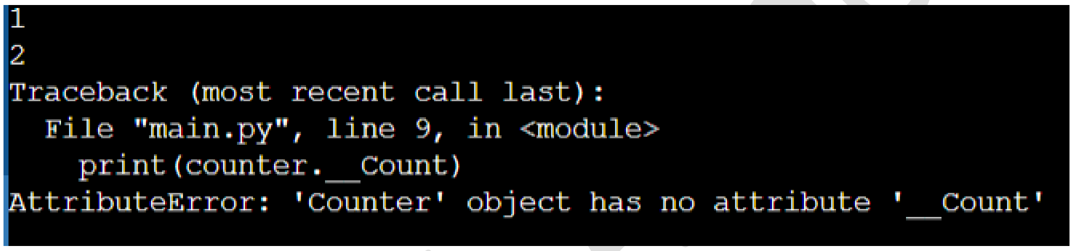
Data hiding isolates the client from a part of the program implementation. We can perform data hiding in Python using the __ double underscore before the prefix. This makes the class members private and inaccessible to the other classes.

Example:

```
    class Counter:

      __Count = 0
      def count(self):
          self.__Count += 1
          print(self.__Count)
    counter = Counter()
    counter.count()
    counter.count()
    print(counter.__Count)
```

output:



```
1
2
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    print(counter.__Count)
AttributeError: 'Counter' object has no attribute '__Count'
```

11.4 Decorators

Using decorators we can add additional functionality to function or method without modifying it.

Example:

```
def check(func):
        def inside(a,b):
                if b==0:
                        print("can not divide by 0")
                        return
                func(a,b)
        return inside
@check
def div(a,b):
    return a/b
print(div(10,0))
```

output:
```
can not divide by 0
None
```

There are some decorators for methods in the class. they are @classmethod,@staticmethod, and @instancemethod.

11.5 Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example of inbuilt polymorphic functions :
```
print(len("python"))
print(len([1,2,3]))
```

Examples of user-defined polymorphic functions :

```
def add(x, y, z = 0):
        return x + y+z
print(add(2, 3))
print(add(2, 3, 4))
```

In python, method overloading is not possible only method overriding is possible. This process of re-implementing a method in the child class is known as **Method Overriding**. It is useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class.

Example:

```
class A:
     def fun1(self):
        print('A1')
    def fun(self):
        print('A')
class B(A):
    def fun(self):
        print('B')
```

```
a=A()
b=B()
a.fun()
b.fun()
b.fun1()
```

output:

```
A
B
A1
```

11.6 Method Resolution Order

The method resolution order controls how the base classes are searched when a method is called in Python. The method or attribute is initially looked for within a class, and then the inheritance order is followed from bottom to top. Class linearization is another name for this structure, and MRO stands for a set of rules (Method Resolution Order). While inheriting from another class, the interpreter needs the means to resolve the methods that are invoked via an instance.

Example:

```
class A:
    def fun(self):
        print('A')
class B():
    def fun(self):
        print('B')
class C(A,B):
    pass
c=C()
c.fun()
```

output:

```
A
```

In the above example, we got A as output because we inherited class A from class C.

Example:

```
class A:
    def fun(self):
        print('A')
class B():
    def fun(self):
        print('B')
class C(B,A):
    pass
c=C()
c.fun()
```

output:

```
B
```

The difference between the first code and the second code is class B is inherited first in the second code that is why we got output B here.

Regular Expression

12.1 match and search method

This method finds a match if it occurs at the start of the string. **re.match()** function of re in Python will search the regular expression pattern and return the first occurrence. The Python RegEx Match method checks for a match only at the beginning of the string. So, if a match is found in the first line, it returns the match object. But if a match is found in some other line, the Python RegEx Match function returns null.

Example:

```
import re
Substring ='string'
String ='''string We are learning regex with geeksforgeeks
         regex is very useful for string matching.
          It is fast too.'''
print(re.match(Substring, String, re.IGNORECASE))
```

output:

```
<re.Match object; span=(0, 6), match='string'>
```

IGNORECASE is to ignore the case sensitivity.

re.search() function will search the regular expression pattern and return the first occurrence. Unlike Python re.match(), it will check all lines of the input string. The Python re.search() function returns a match object when the pattern is found and “null” if the pattern is not found

Example:

```
string="python is interpreted language"
print(re.search("interpreted",string))
```

output:

```
<re.Match object; span=(10, 21), match='interpreted'>
```

There are some flags in python that we can apply in place of re.IGNORECASE those are
re.I is the same as re.IGNORECASE and used for case-insensitive matching.
Re.M is the same as re.MULTILINE and used to ignore newlines in code. After using multiline the code will check each line for string.

re.findall()

Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

Example:

```
import re

string = """Hello my Number is 123456789 and

                    my friend's number is 987654321"""

regex = '\d+'

match = re.findall(regex, string)

print(match)
```

output:

```
['123456789', '987654321']
```

re.sub(*pattern*, *repl*, *string*, *count=0*, *flags=0*)¶

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in a *string* with the replacement *repl*. If the pattern isn't found, the *string* is returned unchanged.

re.split(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in the *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

Database

13.1 SQLite3 Execution

Sqlite3 is used for connecting the database to the python file and getting data from the database.to use sqlite3 we should import the "sqlite3" library.

The syntax for connecting to the database:

sqlite3.connect(database_name)

the cursor will create a cursor for processing database queries.

The syntax for creating a cursor:

connection.cursor()

execute method executes SQL query. If we want to execute many queries we use the excutemany command.

Connection.excute("select * from table");

Connection.commit()

Connection.close()

To commit all changes made in the database we use a commit statement.

To close the database connection we use the close command.

13.2 Basic of Tkinter implementation

Tkinter is used for user interface.for usingTkinterr we should importthe tkinter library.
Simple code to create window in tkinter:

```
from tkinter import *
window=Tk()
# add widgets here

window.title('Hello Python')
window.geometry("300x200")
window.mainloop()
```

opening file dialoag using tkinter:

```
from tkinter import filedialog as fd
Code language: Python (python)
filename = fd.askopenfilename()
```

13.3 Code with Exception

Networking

14.1 Socket Basics

Socket programming is used to connect two nodes or devices in network for communication.we can use socket library for creating simple socket.
Syntax for creating socket:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

AF_INET refers to the address-family ipv4. The SOCK_STREAM means connection-oriented TCP protocol.

Finding ip address using socket

```
import socket

ip = socket.gethostbyname('www.google.com')
print ip
```

s.connect((host_ip, port)) this is code for connecting a ip address. Here host_ip is hosts ip address and port is port number.

14.2 Client and Server Message Sharing Code

```
import socket
host = 'local host'
port = 5000

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(('', port))
s.listen(1)
c, addr = s.accept()
print("CONNECTION FROM:", str(addr))
c.send(b"HELLO, How are you ? \
        Welcome to Akash hacking World")

msg = "Bye............."
c.send(msg.encode())
c.close()
```

Multi-Threading

15.1 Multithreading Basics

Multithreading is defined as the ability of a processor to execute multiple threads concurrently. we can run threads by importing the threading library. Syntax for threads:

t1 = threading.Thread(target=function_name,args=(10,))
for starting a thread we use start function.
t1.start()
Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop the execution of the current program until a thread is complete, we use the **join** method.

t1.join()

15.2 Difference between Multithreading and Multiprocessing

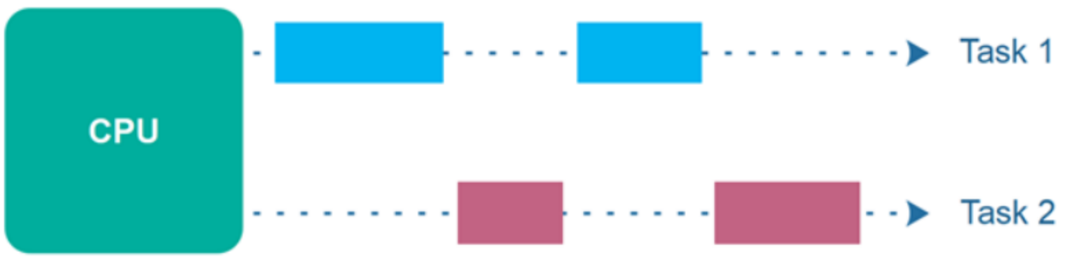
Multithreading:

In Multiprocessing, CPUs are added for increasing computing power.
In Multiprocessing, Many processes are executed simultaneously.
Multiprocessing are classified into Symmetric and Asymmetric.
In Multiprocessing, Process creation is a time-consuming process.
In Multiprocessing, every process owned a separate address space.

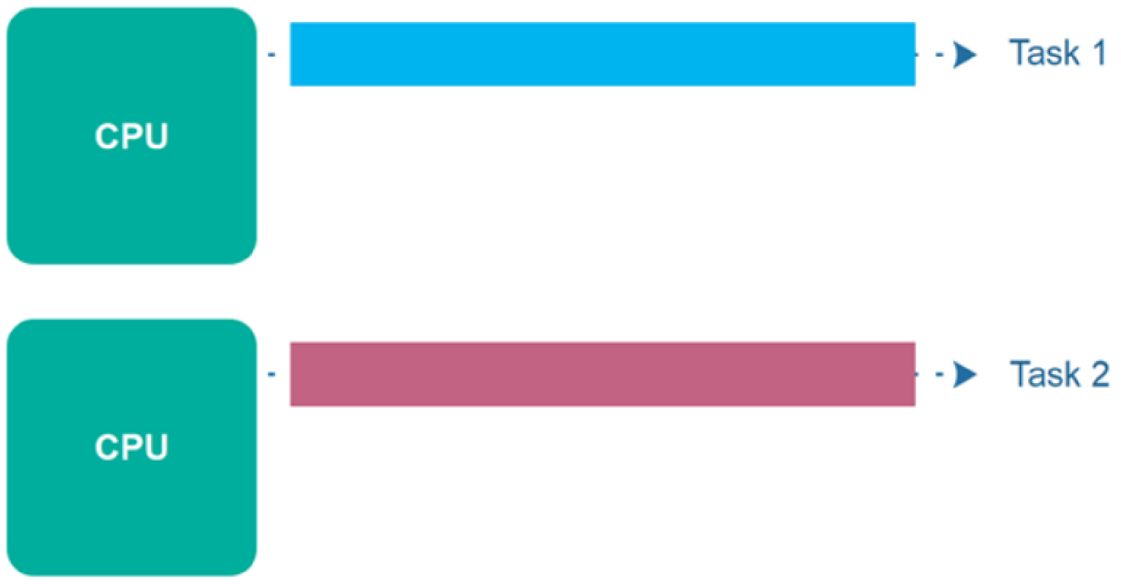
Multiprocessing:
In Multithreading, many threads are created of a single process for increasing computing power.
In multithreading, many threads of a process are executed simultaneously.
Multithreading is not classified into any category.
In Multithreading, process creation is economical.
In Multithreading, a common address space is shared by all the threads.

15.3 Concurrency and Parallel Execution Thread

Concurrency means that an application is making progress on more than one task - at the same time or at least seemingly at the same time.



Parallel execution is when computer uses more than on CPU. It will run more than one task concurrently.



File Operation (txt file)

16.1 Context Manager

Context managers allow you to allocate and release resources precisely when you want to. The most widely used example of context managers is the with a statement. Suppose you have two related operations which you'd like to execute as a pair, with a block of code in between. Context managers allow you to do specifically that. For example:

```
with open('some_file', 'w') as opened_file:
    opened_file.write('Hello!')
```

opening a file:

syntax:

```
File_object = open(r"File_Name","Access_Mode")
```

Access modes for different functions:

Read Only (‘r’): Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.

Write Only (‘w’): Open the file for writing. For the existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist

Append Only (‘a’): Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

For reading data from a text file we use three functions

read:
syntax:
fileobject.read([n])

for n reads at most n bytes. If we did not provide n it will read an entire text file

readline:
syntax:
fileobject.readline([n])

for n it reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

readlines:
syntax:
it reads all line

the syntax for writing data to a text file:

fileobject.write(“text here”)

tell() is used to return the current file position after reading the first line.

Seek() is used to change the current file position to 4, and return the rest of the line.

Date Time Manipulation

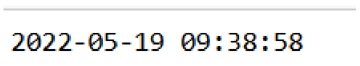
17.1 import datetime and time

For importing datetime we use “import datetime”
For importing time we use “import time”

17.2 change time format

datetime.strftime(format) is syntax to change format of date and time.
Example:

```
from datetime import datetime
now = datetime.now()
date_time = now.strftime("%Y-%m-%d %H:%M:%S")
print(date_time)
```



```
2022-05-19 09:38:58
```

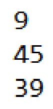
17.3 Adding Hours, minutes and Seconds

We use hour for adding hours and minute for minute and second for adding seconds.

Example:

```
from datetime import datetime
now = datetime.now()
print(now.hour)
print(now.minute)
print(now.second)
```

output:



```
9
45
39
```

Import and Packages

18.1 Path Reference

For getting the path we should import a library “sys”.

sys. path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

When a module(a module is a python file) is imported within a Python file, the interpreter first searches for the specified module among its built-in modules. If not found it looks through the list of directories(a directory is a folder that contains related modules) defined by **sys. path**.