Name: Kartik Bhutada

Roll No: PC09

Panel C

Python Assignment No.6

A. Write a Python program to implement the following concepts

1. Class & Object

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state. An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with actual values.

The self method

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it. If we have a method that takes no arguments, then we still have to have one argument. This is similar to this pointer in C++ and this reference in Java.

__init__ method

The **init** method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

In [1]:
```
# Class for Cars
class Car:

    # Class Variable
    ftype = "petrol"

    # The init method or constructor
    def __init__(self, company, color):

        # Instance Variable
        self.company = company
        self.color = color

# Objects of Dog class
car1 = Car("Mecedes", "black")
```

```
car2 = Car("Volkswagen", "blue")

print('Details of Car1:')
print('Car Type: ', car1.ftype)
print('Company: ', car1.company)
print('Color: ', car1.color)

print('\nDetails of Car2:')
print('Car Type: ', car2.ftype)
print('Company: ', car2.company)
print('Color: ', car2.color)

# Class variables can be accessed using class name also
print("\nAccessing class variable using class name")
print(Car.ftype)
```

```
Details of Car1:
Car Type:  petrol
Company:  Mecedes
Color:  black

Details of Car2:
Car Type:  petrol
Company:  Volkswagen
Color:  blue

Accessing class variable using class name
petrol
```

In [ ]:

2. Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

In [3]:

```
# Base class or super class
class Person(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

    # To check if this person is a student
    def isStudent(self):
        return False
```

```
# Inherited class or Subclass
class Student(Person):

    # Here we return true
    def isStudent(self):
        return True


per = Person("Kartik")  # An Object of Person
print(per.getName(), per.isStudent())

stu = Student("Melissa") # An Object of Student
print(stu.getName(), stu.isStudent())
```

```
Rohan False
Melissa True
```

In [ ]:

3. Operator Overloading

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

Overloading the + operator

In [5]:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x, y)


p1 = Point(1, 2)
p2 = Point(2, 3)

print(p1 + p2) # Python calls p1.__add__(p2) which in turn is Point.__add__(p1,p2)
```

```
(3,5)
```

Overloading relational operators

In [6]:

```
# Overloading the < operator
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
```

```
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x, self.y)

    def __lt__(self, other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1 < p2)
print(p2 < p3)
print(p1 < p3)
```

```
True
False
False
```

In [ ]:

4. Polymorphism

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

Polymorphism in + operator

For integer data types, + operator is used to perform arithmetic addition operation. Similarly, for string data types, + operator is used to perform concatenation.

In [7]:
```
num1 = 753
num2 = 691
print(num1 + num2)

str1 = "Good"
str2 = "Afternoon"
print(str1 + " " + str2)
```

```
1444
Good Afternoon
```

Function Polymorphism

There are some functions in Python which are compatible to run with multiple data types.

One such function is the len() function.

In [8]:
```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

```
9
3
```

2

Class Polymorphism

We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

In [9]:
```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")


class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")


cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Method overriding

In [10]:
```
from math import pi


class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
        pass

    def fact(self):
        return "I am a two-dimensional shape."
```

```
    def __str__(self):
        return self.name


class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length

    def area(self):
        return self.length**2

    def fact(self):
        return "Squares have each angle equal to 90 degrees."


class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return pi*self.radius**2


a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

```
Circle
I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985
```

In [ ]:

5. Error and Exceptions

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when the some internal events occur which changes the normal flow of the program.

Two types of Errors occur in python.

- Syntax errors
- Logical errors (Exceptions)

Syntax Errors

In [11]:
```
if a < 3
```

```
  File "<ipython-input-11-3e28e520013d>", line 1
    if a < 3
```

```
            ^
SyntaxError: invalid syntax
```

In [12]:
```
a =
print(a)
```

```
  File "<ipython-input-12-adf522bb0bd4>", line 1
    a =
      ^
SyntaxError: invalid syntax
```

In [13]:
```
for i in range 0, 10:
    print(i)
```

```
  File "<ipython-input-13-caf26ab33fb9>", line 1
    for i in range 0, 10:
                   ^
SyntaxError: invalid syntax
```

Logical Errors

IndexError

In [15]:
```
names = ("Mike", "Jim", "Oscar", "Angela")

print(len(names))
```

```
4
```

In [16]:
```
print("The fifth member is: ", names[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-16-1aad5c370882> in <module>
----> 1 print("The fifth member is: ", names[4])

IndexError: tuple index out of range
```

ModuleNotFoundError

In [17]:
```
import nonexistent_module
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-17-696f280e00ee> in <module>
----> 1 import nonexistent_module

ModuleNotFoundError: No module named 'nonexistent_module'
```

KeyError

In [18]:
```
companies = {'IBM': 25, 'Intel': 5, 'Cisco': 30, 'Deloitte': 7}
```

In [19]:
```
print("Intake of Microsoft: ", companies['Microsoft'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-19-82dadadb907c> in <module>
----> 1 print("Intake of Microsoft: ", companies['Microsoft'])

KeyError: 'Microsoft'
```

ImportError

In [20]:
```
from math import imaginary_function
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-20-254a4250e411> in <module>
----> 1 from math import imaginary_function

ImportError: cannot import name 'imaginary_function' from 'math' (unknown location)
```

StopIteration

In [23]:
```
li = [1, 2, 3]
it = iter(li)
```

In [24]:
```
next(it)
```

Out[24]: 1

In [25]:
```
next(it)
```

Out[25]: 2

In [26]:
```
next(it)
```

Out[26]: 3

In [27]:
```
next(it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-27-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

TypeError

In [28]:
```
s = "28" + 15
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-45e79fd72a86> in <module>
----> 1 s = "28" + 15

TypeError: can only concatenate str (not "int") to str
```

ValueError

In [29]:
```
print(int("Oops"))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-839299d7537f> in <module>
----> 1 print(int("Oops"))

ValueError: invalid literal for int() with base 10: 'Oops'
```

NameError

In [30]:
```
print(non_existent)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-2202689c8bd2> in <module>
----> 1 print(non_existent)

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

ZeroDivisionError

In [31]:
```
print(65 / 0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-31-b7417cd45cce> in <module>
----> 1 print(65 / 0)

ZeroDivisionError: division by zero
```

In [ ]:
```

```

File Operations

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. Python treats file differently as text or binary and this is important.

Read

In [33]:
```
file = open('Info.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)

file.close()
```

```
Hi my name is Siddharth

I am 21 years old

I live in Pune
```

Write

In [36]:
```
file = open('Info.txt','w')
file.write("This is the write command\n")
file.write("It allows us to write in a particular file")
file.close()
```

In [37]:
```
file = open('Info.txt', 'r')

for each in file:
    print(each)

file.close()
```

```
This is the write command

It allows us to write in a particular file
```

Open

Opening a file refers to getting the file ready either for reading or for writing. This can be done using the open() function. This function returns a file object and takes two arguments, one that accepts the file name and another that accepts the mode(Access Mode). Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

- **Read Only ('r'):** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exist, raises I/O error. This is also the default mode in which the file is opened.
- **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
- **Write Only ('w'):** Open the file for writing. For 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.
- **Write and Read ('w+'):** Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
- **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.
- **Append and Read ('a+'):** Open the file for reading and 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.

Syntax: File_object = open(r"File_Name", "Access_Mode")

The open() function has been demonstrated in the read and write operations.

Close

Though Python automatically closes a file if the reference object of the file is allocated to another file, it is a standard practice to close an opened file as a closed file reduces the risk of being unwarrantedly modified or read.

Python has a close() method to close a file. The close() method can be called more than once and if any operttion is performed on a closed file it raises a ValueError.

In [38]:
```
file = open("Info.txt")

# Reading from file
print(file.read())
```

```
# closing the file
file.close()
```

```
This is the write command
It allows us to write in a particular file
```

In [39]:
```
file.read()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-f3fc120c03c1> in <module>
----> 1 file.read()

ValueError: I/O operation on closed file.
```

Delete

To delete a file, you must import the OS module, and run its os.remove() function:

In [40]:
```
import os

if os.path.exists("Info.txt"):
  os.remove("Info.txt")
else:
  print("The file does not exist")
```

In [41]:
```
file = open("Info.txt", "r")

# Reading from file
print(file.read())

# closing the file
file.close()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-41-d317f793220a> in <module>
----> 1 file = open("Info.txt", "r")
      2
      3 # Reading from file
      4 print(file.read())
      5

FileNotFoundError: [Errno 2] No such file or directory: 'Info.txt'
```

In [ ]:
```

```