OOPS Concept

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.

- **class:** blueprint/template, no memory allocation
- **object:** physical entity of class, memory will be allocated, instance of a class

Why should we use classes?

Classes allow us to logically group our data and functions in a way that is easy to reuse and also easy to build upon if needed.

- Data and functions associated with a specific class are known as attributes and methods respectively.

In [1]:
```
# Create a Class: To create a class, use the keyword class:

class Student:
    x = 5

print(Student)
```

```
<class '__main__.Student'>
```

In [2]:
```
# Create Object: Now we can use the class named Student to create objects:

class Student:
    x = 5

obj1 = Student()
print(obj1)
print(obj1.x)
```

```
<__main__.Student object at 0x7fb2d4c54a30>
5
```

constructor: special method

- name: **init**
- explicitly called
- has atleast one argument (self)
- default constructor generated incase of not defined by user
- to declare and initialize instance variables
- constructor overloading is not avilable in python

The init() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications. To understand the meaning of classes we have to understand the built-in **init**() function.

All classes have a function called **init**(), which is always executed when the class is being initiated.

Use the **init**() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

The **init**() function is called automatically every time the class is being used to create a new object.

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class. Whenever we create methods within a class, they recieve instance (self) as the first argument.

- similar to this keyword: to refer current object
- self is not a keyword in python
- use self within the class to refer instance variable
- PVM provides value of self argument internally

In [ ]:
```
# Example of class in other languages

class Employee
{
    String name;
    int sal;
    static String uni_name= "UPES";
    Employee()
    {
        this.name="Student1"
    }
}
```

In [3]:
```
# class definitions cannot be empty, but if you for some reason have a class definition with no
# put in the pass statement to avoid getting an error.

class Employee:
    pass

emp1 = Employee()
emp2 = Employee()
print(emp1, emp2)
```

```
<__main__.Employee object at 0x7fb2d4c59f40> <__main__.Employee object at 0x7fb2d4c59bb0>
```

In [4]:

```
class Employee:
    pass

emp1 = Employee()
emp2 = Employee()

emp1.firstname = "Ram"
emp1.lastname = "Sharma"
emp1.age = 47

emp2.firstname = "Arjun"
emp2.lastname = "Malhotra"
emp2.age = 36

print(emp1.firstname)
print(emp2.firstname)
```

```
Ram
Arjun
```

In [5]:
```
class Employee:

    def __init__(self, name, eno, salary, address):
        self.name= name
        self.eno= eno
        self.salary= salary
        self.address= address

    def info(self):
        name = 'Shyam'
        print(f'emp name is {name}')
        print(f'emp name is {self.name}')
        print(f'emp no is {self.eno}')
        print(f'emp salary is {self.salary}')
        print(f'emp address is {self.address}')

e1= Employee('Ram', 12, 34546, 'asdf')
e1.info()
```

```
emp name is Shyam
emp name is Ram
emp no is 12
emp salary is 34546
emp address is asdf
```

In [6]:
```
e1= Employee('Ram', 12, 30000, 'ABC')
e2= Employee('Arjun', 34, 54645, 'XYZ')
e3= Employee('Sita', 64, 78999, 'HIJ')
```

In [7]:
```
e3.info()
```

```
emp name is Shyam
emp name is Sita
emp no is 64
emp salary is 78999
emp address is HIJ
```

In [8]:
```
# constructor overloading
class Try:
    def __init__(self):
        print(f'id of self1 is {id(self)}')

    def __init__(self, name):
        print(f'id of self2 is {id(self)}')
        print('hello')

    def __init__(self, name, no):
        print(f'id of self3 is {id(self)}')
        print('hello')
```

In [9]:
```
t1 = Try()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-d31637f9636b> in <module>
----> 1 t1 = Try()

TypeError: __init__() missing 2 required positional arguments: 'name' and 'no'
```

In [10]:
```
t2 = Try('name')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-22c4a42ed102> in <module>
----> 1 t2 = Try('name')

TypeError: __init__() missing 1 required positional argument: 'no'
```

In [11]:
```
t3 = Try('name','no')
```

```
id of self3 is 140406050655776
hello
```

In [12]:
```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age

    def myfunc(abc):
        print("Hello my name is " + abc.name)
        print("My age is: " + str(abc.age))

p1 = Person("John",36)
p1.myfunc()
```

```
Hello my name is John
My age is: 36
```

In [13]:
```
# Modify Object Properties: You can modify properties on objects like this:
p1.age = 40
p1.myfunc()
```

```
Hello my name is John
My age is: 40
```

In [14]:
```
# Delete Object Properties: You can delete properties on objects by using the del keyword
del p1.age
p1.myfunc()
```

```
Hello my name is John

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-ca0a7710e80e> in <module>
      1 # Delete Object Properties: You can delete properties on objects by using the del keyword
      2 del p1.age
----> 3 p1.myfunc()

<ipython-input-12-0dde5f2fe44f> in myfunc(abc)
      6     def myfunc(abc):
      7         print("Hello my name is " + abc.name)
----> 8         print("My age is: " + str(abc.age))
      9
     10 p1 = Person("John",36)

AttributeError: 'Person' object has no attribute 'age'
```

In [15]:
```
# You can delete objects by using the del keyword
del p1
print(p1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-64abcf942e76> in <module>
      1 # You can delete objects by using the del keyword
      2 del p1
----> 3 print(p1)

NameError: name 'p1' is not defined
```

In [16]:
```
# class gives a blue-print of the object
class Student: # a class having no attributes and no methods
    pass

#creating two different objects
stud1 = Student()
stud2 = Student()

print(stud1, stud2) #Two different class objects
```

```
<__main__.Student object at 0x7fb2d4c207f0> <__main__.Student object at 0x7fb2d4c206d0>
```

In [17]:
```
class Student:
    def __init__(self, fname, lname, sapid): # think of this like a constructor
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'

    def fullname(self):
        return f'{self.fname} {self.lname}'

stud1 = Student('Ram', 'Gupta', '50006704') #stud1 instance is passed to self by default

print(stud1.email)
print(stud1.fullname())

print(Student.fullname(stud1)) #calling method using class and passing instance
```

```
50006704@stu.upes.ac.in
Ram Gupta
Ram Gupta
```

- instance: object level
- static: class level
- local: inside block (for temporary requirement)

Types of variables: instance, static/class

Instance Variables

- Contain data unique to an instance.
- Varies from object to object

- Create instance variable using:

1. within class---create and initialize using construtor
2. within class---Inside instance method by using self
3. outside class---using object reference

- Access instance variable using:

1. within class---by using self
2. outside class-- using obj ref variable

- delete instance variable using:

1. within class---del self.variable name
2. outside class-- del ref name

In [18]:
```python
# Example of instance variables
class Student:
    def __init__(self, fname, lname, sapid):
        # Instance variables
        self.fname = fname                    # names need not be same as arguments
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'

    def fullname(self):
        return(f'{self.fname} {self.lname}')

stud1 = Student('Ram', 'Gupta', '50006704') #stud1 instance is passed to self by default

print(stud1.email)

# ways to call a class method
print(stud1.fullname())        # # calling method using instance
print(Student.fullname(stud1)) # calling method using class and passing instance
```

```
50006704@stu.upes.ac.in
Ram Gupta
Ram Gupta
```

Class/Static Variables

- Variable that are shared among all instances of a class
- Can be accessed trhough class itself or using an instance of the class
- Common for all objects, better memory management

In C++ and Java, we can use static keywords to make a variable a class variable. The variables which don't have a preceding static keyword are instance variables. The Python approach is simple; it doesn't require a static keyword.

When we declare a variable inside a class but outside any method, it is called as class or static variable in python.

- Create static variable using:

1. within class---inside constructor using classname

2. within class---inside instance methods using classname
3. within class---inside class methods using classname or cls variable
4. within class---inside static methods using classname
5. within class---outside all methods
6. outside class-- using class name

- Access static variable using:

1. within class---by using cls, class name and self
2. outside class-- using class name or ref variable

- Modify static variable using:

1. within class---by using cls, class name
2. outside class-- using class name

In [19]:
```
# Change name: can be hidden in multiple methods
# Access the name of university: only through object
# Pass info everytime an instance is created

class Student:
    def __init__(self, fname, lname, sapid, univ):
        # Instance variables
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'
        self.univ = univ

    def fullname(self):
        return(f'{self.fname} {self.lname}')

stud1 = Student('Ram', 'Gupta', '50006704','UPES')
stud2 = Student('Arjun', 'Uppal', '50006765','UPES')

print(stud1.univ)
print(stud2.univ)
```

```
UPES
UPES
```

In [20]:

```
class Student:
    def __init__(self, fname, lname, sapid):
        # Instance variables
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'

    def univname(self):
        self.univ = 'UPES'
        return(f'{self.univ}')

stud1 = Student('Ram', 'Gupta', '50006704')
stud2 = Student('Arjun', 'Uppal', '50006765')

print(stud1.univname())
print(Student.univname(stud2))
```

```
UPES
UPES
```

In [21]:
```python
# Example of class variables
class Student:
    # class variable
    univ = 'UPES'

    def __init__(self, fname, lname, sapid):
        # Instance variables
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'

    def fullname(self):
        return(f'{self.fname} {self.lname} {self.univ}')  # Accessing class variable using insta
#         return(f'{self.fname} {self.lname} {Student.univ}')  # Accessing class variable using

stud1 = Student('Ram', 'Gupta', '50006704')
stud2 = Student('Arjun', 'Uppal', '50006765')

# Accessing class variables using instance of a class
print(stud1.univ)
print(stud2.univ)

# Accessing class variables using a class
print(Student.univ)
```

```
UPES
UPES
UPES
```

Note: When we try to access a variable using instance, it first checks if the isntance contains that attribute. If it doesn't, then it will check if the class or any class that it inherits from has that attribute.

Class or static variable are quite distinct from and does not conflict with any other member variable with the same name. Below is a program to demonstrate the use of class or static variable.

In [22]:

```python
# class variables vs instance variables

class Student:

    college = 'UPES' # class variable
    num_of_students = 0

    def __init__(self, fname, lname, sapid):
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.email = sapid + '@stu.upes.ac.in'

        Student.num_of_students += 1

    def fullname(self):
        return f'{self.fname} {self.lname} {self.college}'

stud1 = Student('Arjun', 'Uppal', '50006765')
stud2 = Student('Ram', 'Sharma', '500067827')

print(Student.college, stud1.college, stud2.college) # two ways to access class variable
print()

print(stud1.__dict__) # returns a dict of all instance variables of a particular instance
print(Student.__dict__)# returns a dict of all class variables + info related to class
print()

# changing value of class variable: changes for the entire class, i.e. all instances
Student.college = 'NIT'
print(Student.college, stud1.college, stud2.college)
print(Student.__dict__)
print()

# changing value of class variable using an instance: changes for that specific instance
stud1.college = 'IIT'
print(Student.college, stud1.college, stud2.college)
print(stud1.__dict__) # now college becomes instance variable for stud1 instance
print()

# another use of class variable where there is no sense of using self/instance
print(Student.num_of_students)
```

```
UPES UPES UPES

{'fname': 'Arjun', 'lname': 'Uppal', 'sapid': '50006765', 'email': '50006765@stu.upes.ac.in'}
{'__module__': '__main__', 'college': 'UPES', 'num_of_students': 2, '__init__': <function Student._
__init__ at 0x7fb2d64811f0>, 'fullname': <function Student.fullname at 0x7fb2d4380e50>, '__dict__':
<attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' o
bjects>, '__doc__': None}

NIT NIT NIT
{'__module__': '__main__', 'college': 'NIT', 'num_of_students': 2, '__init__': <function Student.__
init__ at 0x7fb2d64811f0>, 'fullname': <function Student.fullname at 0x7fb2d4380e50>, '__dict__': <
attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' ob
jects>, '__doc__': None}

NIT IIT NIT
{'fname': 'Arjun', 'lname': 'Uppal', 'sapid': '50006765', 'email': '50006765@stu.upes.ac.in', 'coll
ege': 'IIT'}

2
```

Types of methods: instance, static, class

Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have maintenance benefits.

Basis for differentiation	Instance method	Class method	Static method
Decorator	No decorator required.	Decorator "@classmethod" is used	Decorator "@staticmethod" is used
Argument	Takes instance of class as first argument (self).	Takes class as first argument (cls)	class or instance not required to call a static method. These methods are unaware of the class or objects they reside in.
Access	Needs instance to call this method.	This method can be accessed with class name without creating instance.	This method can be accessed with class name without creating instance.
Usage	Typically used to access or update instance variables	Typically used to access or update class variable. Is also used as alternative initializer/constructor.	Typically used to create some utility method which is relevant to, yet independent of the class.

Instance Methods

Instance attributes are those attributes that are not shared by objects. Every object has its own copy of the instance attribute.

Instance methods need a class instance and can access the instance through self.

For example, consider a class shapes that have many objects like circle, square, triangle, etc. having its own attributes and methods. An instance attribute refers to the properties of that particular object like edge of the triangle being 3, while the edge of the square can be 4.

An instance method can access and even modify the value of attributes of an instance. It has one default parameter: self

In [23]:
```
class shape:

    # Calling Constructor
    def __init__(self, edge, color):
        self.edge = edge
        self.color = color

    # Instance Method
    def finEdges(self):
        return self.edge

    # Instance Method
    def modifyEdges(self, newedge):
        self.edge = newedge

circle = shape(0, 'red')
square = shape(4, 'blue')

# Calling Instance Method
print("No. of edges for circle: "+ str(circle.finEdges()))

# Calling Instance Method
square.modifyEdges(6)
print("No. of edges for square: "+ str(square.finEdges()))

```

```
No. of edges for circle: 0
No. of edges for square: 6
```

Class Methods

- Class methods don’t need a class instance. They can’t access the instance (self) but they have access to the class itself via cls.
- @classmethod --> decorator is compulsory

In [24]:

```
class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    def __init__(self,first,last,pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + "@gmail.com"

        Employee.num_of_emps += 1

    def fullname(self):
        return f'{self.first} {self.last}'

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    # Decorators: Alter functionality of a method, receive class as first argument instead of in
    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

emp1 = Employee('Arjun', 'Uppal', '50006765')
emp2 = Employee('Ram', 'Sharma', '500067827')
print(Employee.raise_amt, emp1.raise_amt, emp2.raise_amt)

# same as Employee.raise_amt = 2.05, but we are using a class method
Employee.set_raise_amt(2.05)
print(Employee.raise_amt, emp1.raise_amt, emp2.raise_amt)

# Run class method from instance: changes all instances
emp1.set_raise_amt(3.06)
print(Employee.raise_amt, emp1.raise_amt, emp2.raise_amt)
```

```
1.04 1.04 1.04
2.05 2.05 2.05
3.06 3.06 3.06
```

In [25]:

```
# Using class methods as alternative constructors
# Consider if info of an employee is passed as a single string that needs to be split

class Employee:

    num_of_emps = 0
    raise_amt = 1.04

    def __init__(self,first,last,pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + "@gmail.com"

        Employee.num_of_emps += 1

    def fullname(self):
        return f'{self.first} {self.last}'

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)

    @classmethod
    def set_raise_amt(cls, amount):
        cls.raise_amt = amount

    @classmethod
    def from_string(cls, e_string):
        first, last, pay = e_string.split('-')
        return cls(first, last, pay)   # same as Employee(first, last, pay), will create a new e

emp_str_1 = 'Arjun-Uppal-30000'
first, last, pay = emp_str_1.split('-')
new_emp1 = Employee(first, last, pay)
print(new_emp1.email)

# We don't want user to parse these stings everytime he wants to create a new employee.
# So we create an alternate constructor as class method
emp_str_2 = 'Ram-Goyal-5000'
new_emp2 = Employee.from_string(emp_str_2)

new_emp3 = Employee.from_string('Shyam-Saxena-80000')
```

```
43
44 print(new_emp2.email)
45 print(new_emp3.email)
```

```
Arjun@gmail.com
Ram@gmail.com
Shyam@gmail.com
```

Static Methods

- instance method automatically pass instance as their first arguement as self
- class method automatically pass class as their first arguement as cls
- static method is just a regular function in class which does not use any instance or class itself

Static methods are included in the classes because they have some logical connection to the class

In [26]:

```
# See if a date was a working day or not: has connection to Employee class but does not specific
# on any instance or class variable

class Employee:

    covid_ded = 5
    emp_count=0

    def __init__(self, fname, lname, salary):
        self.fname= fname
        self.lname= lname
        self.email= fname[0]+ '.'+ lname+'@ddn.upes.ac.in'
        self.salary= salary
        Employee.emp_count+=1

    # instance method
    def full_name(self):
        return f'{self.fname} {self.lname}'

    # instance method
    def apply_ded(self):
        self.salary= self.salary - self.salary * (Employee.covid_ded/100)

    @classmethod
    def set_covid_amount(cls, amount):
        cls.covid_ded = amount

    @staticmethod
    # Do not take instance/class as first argument, simple pass the reuired args
    def is_workday(day):
        if day.weekday()== 5 or day.weekday()== 6:
            return True
        else:
            return False

    # represents the class objects as a string – it can be used for classes
    # Use static methods where you do not need to use class or instance
    def __str__(self):
        return f'using __str__: {self.fname} {self.lname} {self.salary}'

e1=Employee('Ram', 'Tiwari', 50000)
print(e1.full_name())
```

```
43
44 e1.apply_ded()
45 print(e1.salary)
46
47 Employee.set_covid_amount(2)
48 print(Employee.covid_ded)
```

```
Ram Tiwari
47500.0
2
```

In [27]:

```
1 # example - datetime.py   https://github.com/python/cpython/blob/master/Lib/datetime.py, line -
2 # https://docs.python.org/3/library/datetime.html#datetime.date
3
4 import datetime
5
6 date = datetime.date(2020, 11, 7)
7 print(Employee.is_workday(date))
```

```
True
```

In [28]:

```
1 e2 = Employee('Shayam', 'Sharma', 60000)
2 print(e2)
```

```
using __str__: Shayam Sharma 60000
```

object.str(self)

If we try to execute following code without **str** method then statement will result in something like following in output window <**main**.Student object at 0x0191F850>
In order to have a meaningful string representation of Student object we can add this method and return a string. You may refer following link for more information

str and repr methods

https://www.educative.io/edpresso/what-is-the-str-method-in-python (https://www.educative.io/edpresso/what-is-the-str-method-in-python)
https://www.journaldev.com/22460/python-str-repr-functions (https://www.journaldev.com/22460/python-str-repr-functions)

In [29]:
```
class Person:

    def __init__(self, personName, personAge):
        self.name = personName
        self.age = personAge

#     def __str__(self):
#         return self.name + ' ' + str(self.age)

p = Person('Pankaj', 34)
print(p)
print(p.__str__())
```

```
<__main__.Person object at 0x7fb2d4c77220>
<__main__.Person object at 0x7fb2d4c77220>
```

Some more examples:

In [30]:
```python
# Example 1:

class Employee:
    college_name= 'UPES'

    def __init__(self, college_name, eno, salary, address):
        self.college_name= college_name
        self.eno= eno
        self.salary= salary
        self.address= address

    def info(self):
        print('*'*20)
        print(f'emp name is {self.name}')
        print(f'emp no is {self.eno}')
        print(f'emp salary is {self.salary}')
        print(f'emp address is {self.address}')
        print('*'*20)

    def diwali_bonus(self, incentive):
        updated_sal= self.salary + self.salary * (incentive/100)
        print(updated_sal)

    @classmethod
    def getCollegeName(cls):
        print(cls.college_name)

    @staticmethod
    def findAvg(x,y):
        print(x+y/2)
```

In [31]:
```python
e1=Employee('DIT', 34, 50000, 'abc' )
print(e1.college_name)
print(Employee.college_name)
```

```
DIT
UPES
```

In [32]:
```
e1.age = 30
print(e1.age)
Employee.college_name= 'DIT'
print(Employee.college_name)
```

```
30
DIT
```

In [33]:
```
e1.diwali_bonus(5)
```

```
52500.0
```

In [34]:
```
Employee.getCollegeName()
```

```
DIT
```

In [35]:
```
Employee.findAvg(3,4)
e1.findAvg(3,4)
```

```
5.0
5.0
```

In [36]:

```
# Example 2:

import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def area(self):
        return self.circle_area(self.radius)

    # Instead of calculating the area directly within area(), using the well-known circle area f
    # we have factored that out to a separate circle_area() static method.
    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi

p = Pizza(4, ['mozzarella', 'tomatoes'])
p.__dict__
p.area()
p.circle_area(4)
```

Out[36]: 50.26548245743669

In [37]:

```python
# Example 3:

# instance, class and static methods
# instance method automatically pass instance as their first arguement as self
# class method automatically pass class as their first arguement as cls
# static method is just a regular func in class which does not use any instance or class itself

class Test:

    subject = 'Python'
    max_marks = 80
    num_of_students = 0

    def __init__(self, fname, lname, sapid, marks):
        self.fname = fname
        self.lname = lname
        self.sapid = sapid
        self.marks = marks
        self.email = sapid + '@stu.upes.ac.in'

        Test.num_of_students += 1

    def result(self):
        return f'{self.fname} {self.lname} has scored {self.marks}/{Test.max_marks}'

    @classmethod
    def change_max_marks(cls, update):
        cls.max_marks = update

    @classmethod
    def from_string(cls, string):
        fname, lname, sapid, marks = string.split()
        return cls(fname, lname, sapid, marks)

    @staticmethod
    def print_test_date(date):
        return 'Date of test - ' +  '/'.join(date.split())

stud1 = Test('Shyam', 'Singh', '5007365', 70)
stud2 = Test('Sita', 'Sharma', '5000627', 75)

# class variables can be updated if called by class
```

```
# class object can change value of class variables using class methods
print(stud1.result())

stud1.change_max_marks(90)
print(Test.result(stud1))

Test.change_max_marks(100)
print(stud1.result())

print(Test.__dict__)
print()

# class method acting as a constructor to parse input before creating an object
stud3 = Test.from_string('Arjun Singh 50006264 65')
print(stud3.result())
print()

# static method
print(Test.print_test_date('24 9 2020'))
```

```
Shyam Singh has scored 70/80
Shyam Singh has scored 70/90
Shyam Singh has scored 70/100
{'__module__': '__main__', 'subject': 'Python', 'max_marks': 100, 'num_of_students': 2, '__init__':
<function Test.__init__ at 0x7fb2d43aa4c0>, 'result': <function Test.result at 0x7fb2d43aa940>, 'ch
ange_max_marks': <classmethod object at 0x7fb2d43b7910>, 'from_string': <classmethod object at 0x7f
b2d43b7160>, 'print_test_date': <staticmethod object at 0x7fb2d43b76d0>, '__dict__': <attribute '__
dict__' of 'Test' objects>, '__weakref__': <attribute '__weakref__' of 'Test' objects>, '__doc__':
None}

Arjun Singh has scored 65/100

Date of test - 24/9/2020
```

Some questions on OOPS concept

Question 1

Create a class Employee and generate email of employee

In [38]:
```
class Employee:

    def __init__(self, fname, lname, salary):
        self.fname = fname
        self.lname = lname
        self.salary = salary
        self.email = fname + '.' + lname + '@company.com'

emp1 = Employee('Mohandas', 'Gandhi', 50000)
print(emp1.email)
```

```
Mohandas.Gandhi@company.com
```

In the previous example add the following methods:

- getEmail : should return the email id

- getFullName : should return full name (first name followed by last name)

- getPay : should return the pay

In [39]:
```
class Employee:

    def __init__(self, fname, lname, salary):
        self.fname = fname
        self.lname = lname
        self.salary = salary
        self.email = fname + '.' + lname + '@company.com'

    def getFullName(self):
        return (self.fname + ' ' + self.lname)

    def getPay(self):
        return (self.salary)

    def getEmail(self):
        return (self.email)

emp_1 = Employee('Mohandas', 'Gandhi', 50000)

print('Full Name: {}'.format(emp_1.getFullName()))
print('Salary: {}'.format(emp_1.getPay()))
print('Email ID: {}'.format(emp_1.getEmail()))
```

```
Full Name: Mohandas Gandhi
Salary: 50000
Email ID: Mohandas.Gandhi@company.com
```

Question 2

List the risk associated with the implementation of Account class. Suggest a solution.

In [40]:
```
class Account:

    def __init__(self, initial_amount):
        self.balance = initial_amount

    def withdraw(self,amount):
        self.balance = self.balance - amount

    def deposit(self,amount):
        self.balance = self.balance + amount

ac = Account(1000)
ac.balance = 2000
ac.balance = -1000
print(ac.balance)
```

```
-1000
```

In this case, the balance can be changed randomly by the class user and can also be set to a non-permissable value (like -1000).

To solve this, we can call methods intead of updating the balance.

In [41]:
```
class Account:

    def __init__(self, initial_amount):
        self.balance = initial_amount

    def withdraw(self,amount):
        self.balance = self.balance - amount

    def deposit(self,amount):
        self.balance = self.balance + amount

ac = Account(1000)
ac.deposit(2000)
ac.withdraw(1000)
print(ac.balance)

# We can also make balance a private variable, but then it would just update the balance directl
# and no use of methods is needed.
```

```
2000
```

Question 3

A dog trainer had two dogs: Fido and Buddy. Fido was trained a trick of “roll over” and Buddy was learned “play dead”. Is the code written correctly to represent this situation?

- A. Yes

  ```
  prove by printing print(d.tricks) and print(e.tricks)
  ```

- B. No

  ```
  if no, then rewrite the code
  ```

In [42]:
```
class Dog:
    tricks = []

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

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
print(e.tricks)
```

```
['roll over', 'play dead']
['roll over', 'play dead']
```

In [43]:
```
# No, the code is not correct. It appends the trick name to the class variable since list is a m
# type.
# We can make 'tricks' an instance variable because each dog will have its own tricks.

class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
print(d.tricks)
print(e.tricks)
```

```
['roll over']
['play dead']
```

Question 4

Write logic for from_string method such that it becomes alternative constructor; meaning it should create an object of Employee at #stmt1 with first name last name and pay values from emp_1_str string

In [44]:
```
class Employee:

    @classmethod
    def from_string(cls,emp_str):
        firstname,lastname,pay = emp_str.split('-')
        return cls(firstname,lastname,pay)

    def __init__(self,first,last,pay):
        self.firstname = first
        self.lastname = last
        self.pay = pay

emp_1_str = 'John-Abraham-50000'
emp_1 = Employee.from_string(emp_1_str)        # stmt1
print(emp_1.firstname)
print(emp_1.lastname)
print(emp_1.pay)
```

```
John
Abraham
50000
```

Question 5

Requirement: Both the counters (counter1 & counter2) in following code, access the same __item_count from Store. User can get the number of items in store by calling getItemCount method.

counter1 = Store()
counter2 = Store()
#add 2 items to store from counter1
#issue 1 item at counter1
#getItemCount in the Store

Provide body for the 3 methods. Logic is as follows:

- addItem (count): __item_count += count
- issueItem (count): __item_count -= count
- getItemCount(): returns __item_count

Justify method type (instance or static or class) for each method. Test your logic for above requirement.

In [45]:
```
# We cannot access private variable directly. Hence classmethod is used to change the no. of ite

class Store:
    __item_count = 100

    #adds to count to __item_count
    @classmethod
    def addItem(cls,count):
        cls.__item_count += count

    #subtracts count from __item_count
    @classmethod
    def issueItem(cls,count):
        cls.__item_count -= count

    #returns __item_count
    @staticmethod
    def getItemCount():
        return (Store.__item_count)

Store.addItem(10)
counter1 = Store()
counter2 = Store()
counter1.addItem(2)
counter1.issueItem(1)
print(Store.getItemCount())
```

```
111
```

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the

object's address in memory. The 'is' operator compares the identity of two objects; the id() function returns an integer representing its identity.

In [46]:
```
# This code shows you that everything in Python is indeed an object.
# Each object contains at least three pieces of data:

# Reference count: The reference count is for memory management
# Type: The type is used at the CPython layer to ensure type safety during runtime.
# Value: Finally, there's the value, which is the actual value associated with the object.

print(isinstance(2, object))
print(isinstance(list(), object))
print(isinstance(True, object))
```

```
True
True
True
```

1. Python does not have variables. It has names. Yes, this is a pedantic point, and you can certainly use the term variables as much as you like. It is important to know that there is a difference between variables and names.

   Note: The PyObject is not the same as Python's object. It's specific to CPython and represents the base structure for all Python objects.

2. PyObject is defined as a C struct, so if you're wondering why you can't call typecode or refcount directly, its because you don't have access to the structures directly. Method calls like sys.getrefcount() can help get some internals.

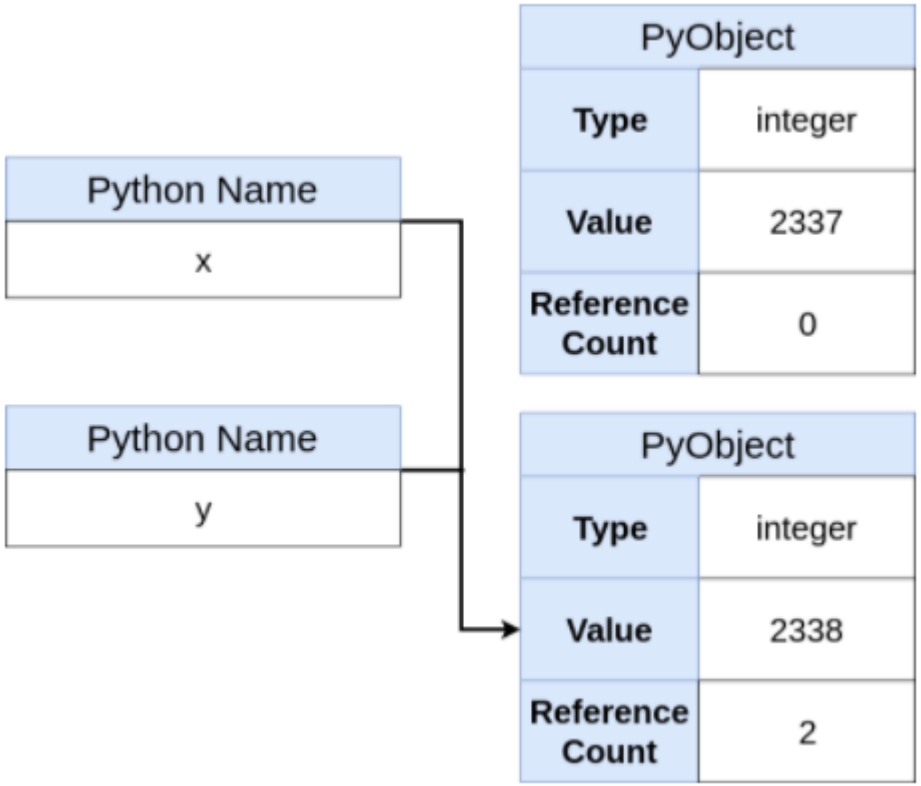
   Now you can see that a new Python object has not been created, just a new name that points to the same object. Also, the object's refcount has increased by one. You could check for object identity equality to confirm that they are the same:

3. Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether

Implementation note: the current implementation uses a reference-counting scheme which collects most objects as soon as they become unreachable, but never collects garbage containing circular references

In [47]:
```
x = 2337
x = 2338
y = x
y is x
```

Out[47]: True



In [48]:
```
# for every class, pvm creates one class class object to hold class level info
# refer static data using cls varibale

x = 5
id(x)
```

Out[48]: 11428384

In [ ]:
```
1
```