Introduction to PYTHON

Object Oriented Concepts

By: Atul Kumar Uttam

Need of OOP

- Let’s say you wanted to track employees in an organization.
- You need to store some basic information about each employee,
  - name,
  - age,
  - position,
  - Year of joining.

Need of OOP

kirk = ["James Kirk", 34, "Captain", 2265]

spock = ["Spock", 35, "Science Officer", 2254]

mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]

Need of OOP

kirk[0] ……..?
mccoy[0] ………….?
spock [0] …………..?

Kirk[1] ………….?
mccoy[1] ………….?
spock [1] ………….?

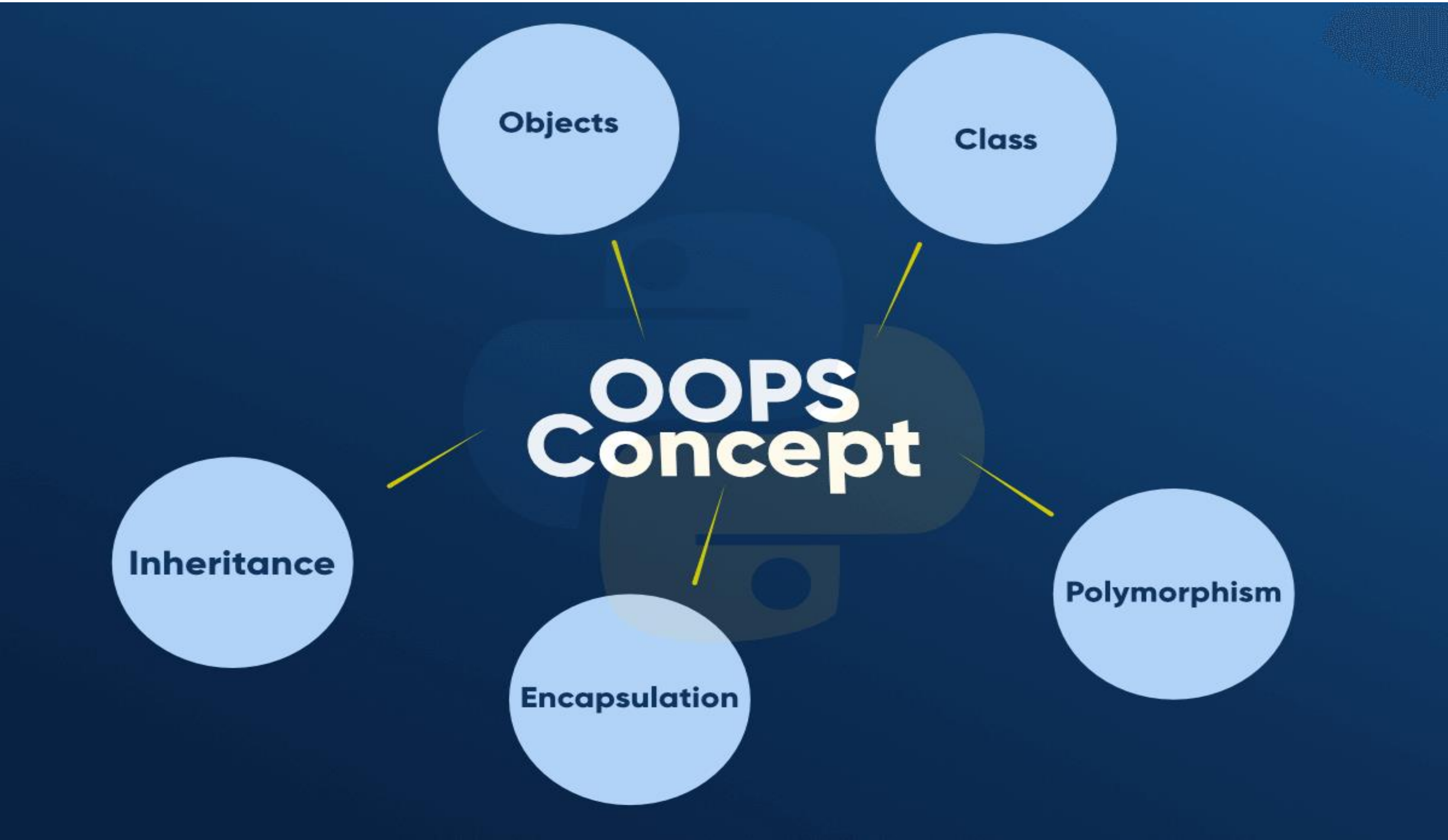
- Classes are used to create user-defined data structures.

- Classes also have special functions, called **methods**, that define behaviors and actions that an object created from the class can perform with its **data**.

Object-Oriented Programming

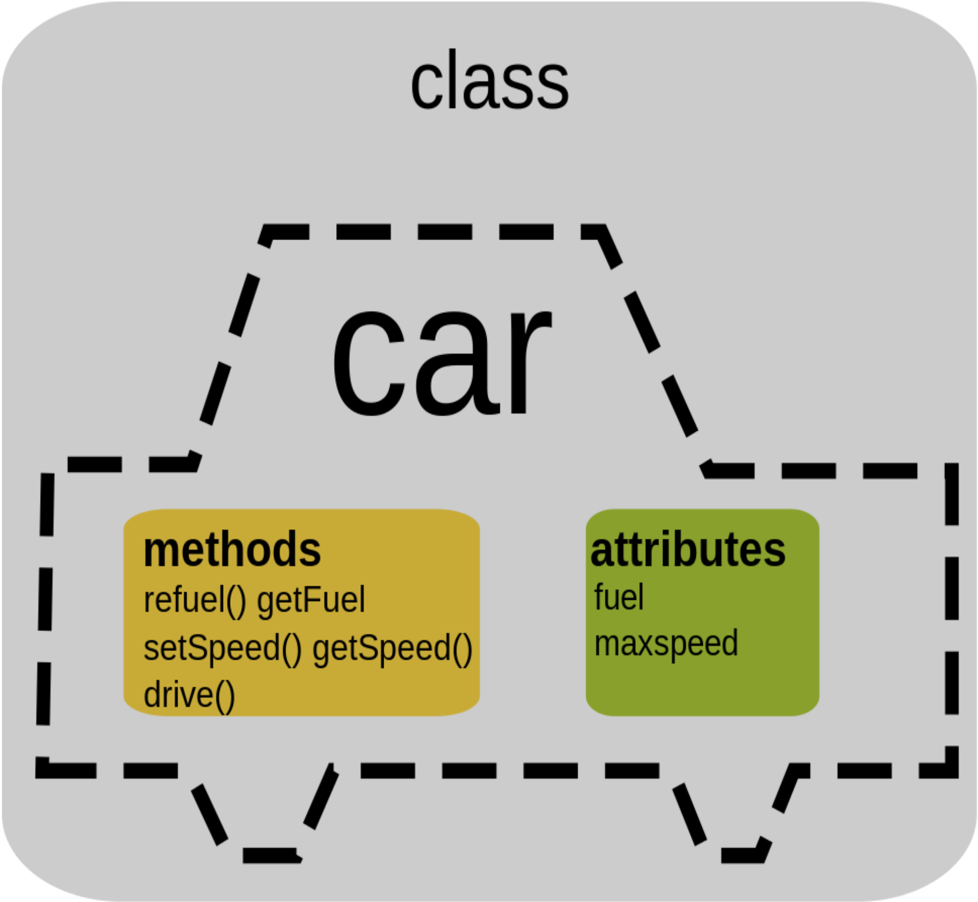
- The object is related to real-word entities such as book, house, pencil, etc.
- The oops concept focuses on writing the reusable code.
- It is a widespread technique to solve the problem by creating objects.
- The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

Object-oriented programming Concepts



Class

- A class is a collection of objects.
- A class contains the blueprints or the prototype from which the objects are being created.
- It is a logical entity that contains some attributes and methods.



Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
- Eg.: Myclass.Myattribute

Class Definition Syntax

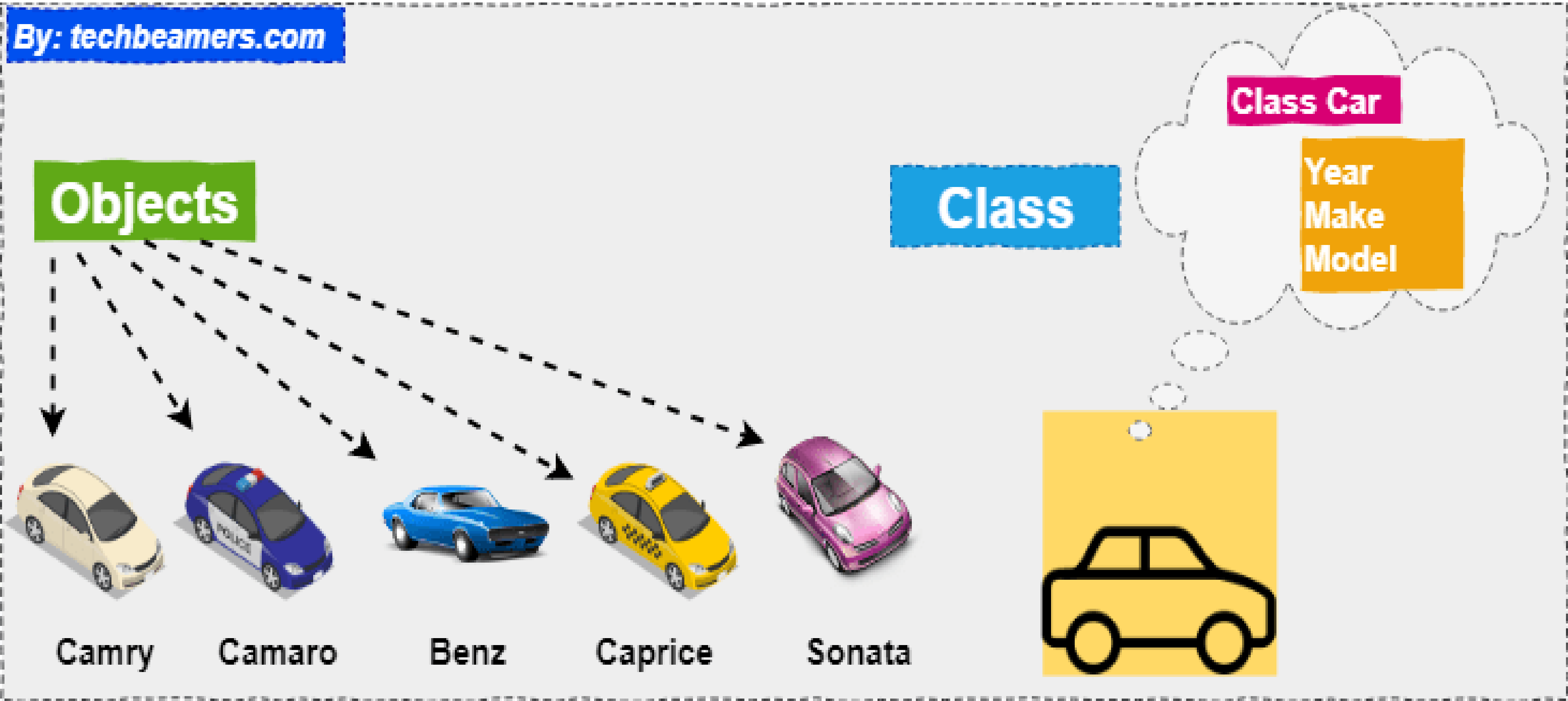
```
class   ClassName:
        # Statement-1
        . . .
        # Statement-N
```

```python
# Python3 program to demonstrate
# defining a class

class Car:
        pass
```

Objects

- The object is an entity that has a state and behavior associated with it.



```python
class Dog:
    pass
```

An object consists of :

- **Identity:**
  - It gives a unique name to an object and enables one object to interact with other objects.
- **State:**
  - It is represented by the attributes of an object.
  - It also reflects the properties of an object.
- **Behavior:**
  - It is represented by the methods of an object.
  - It also reflects the response of an object to other objects.

Identity	**State/Attributes**	**Behaviors**
Name of dog	*Breed*	*Bark*
	Age	*Sleep*
	Color	*Eat*

```python
class Dog:
    pass
```

#Creating an object

obj = Dog()

Some Basic Keywords

self

- 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.

- When we call a method of this object as

  myobject.method(arg1, arg2)

- this is automatically converted by Python into

  MyClass.method(myobject, arg1, arg2)

__init__ method

- The is similar to constructors in C++ and Java.
- It is run 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.

```
# A Sample class with init method
class Car:
    # init method or constructor
    def __init__(self):
        print(“Car Object is created”)

    # Sample Method
    def start_engine(self):
        print(‘Car has been started’)


c1 = Car()
c1.start_engine()
```

Output:
Car Object is created
Car has been started

```
# A Sample class with init method
  class Person:
      # init method or constructor
      def __init__(self, name):
          self.name = name

      # Sample Method
      def say_hi(self):
          print('Hello, my name is',self.name)


 p = Person('James')              #  Person.__init__(P, 'James')

p.say_hi()                        #   Person.say_hi(P)
```

Output:
Hello, my name is James

Class and Instance Variables

- **Instance variables** are for data, unique to each instance
- **Class variables** are for attributes and methods shared by all instances of the class.
- **Instance variables** are variables whose value is assigned inside a constructor or method with self.
- **Class variables** are variables whose value is assigned in the class.

```python
class Dog:
    # Class Variable
    animal = 'pet dog'
    def __init__(self, breed, color):
        # Instance Variable
        self.breed = breed
        self.color = color
```

```python
# Objects of Dog class
Rodger = Dog("Pug", "brown")
Buzo = Dog("Bulldog", "black")

print('Rodger is a', Rodger.animal)
print('Breed: ', Rodger.breed)
print('Color: ', Rodger.color)

print('Buzo is a', Buzo.animal)
print('Breed: ', Buzo.breed)
print('Color: ', Buzo.color)

# Class variables can be accessed using
# class name also

print(Dog.animal)
```

```
class  Dog:
    # Class Variable
     animal = 'pet dog'
     def __init__(self, breed):
          # Instance Variable
          self.breed = breed
    # Adds an instance variable
     def  setColor(self, color):
           self.color = color
     # Retrieves instance variable
     def getColor(self):
            return  self.color
```

```
Rodger = Dog("pug")
Rodger.setColor("brown")
print(Rodger.getColor())
```

```
class Dog:
    attr1 = “pet dog"
    def __init__(self, name):
        self.name = name
    def speak(self):
        print(“hello”)


# Object instantiation
dog1 = Dog("Rodger")


# Accessing class methods
print(dog1.speak())
```

Example of class variable

```
class student:
    count=0                                   # class variable
    def __init__(self, na, ma):
        print("Constructor invoked")
        self.name=na
        self.marks=ma
        student.count = student.count+1
    def display(self):
        print("NAME ",self.name)
        print("MARKS ",self.marks)
    def total(self):
        print("Total students ", student.count)
```

```
a=student("abc",99)        #Constructor invoked
b=student("xyz",88)        #Constructor invoked

a.display()                #NAME  abc
                           #MARKS  99

b.display()                #NAME  xyz
                           #MARKS  88

student.count              # 2
a.count                    # 2
b.Count                    # 2
```

- The variable count is a **class variable whose value is shared among all the instances of the class**.
- This can be accessed as **student.count** from inside the class or outside the class.
- The first method **__init__()** is a special method, which is called initialization method that Python calls when you create a new instance of this class.

Destructors in Python

```
# Python program to illustrate destructor
class  Employee:

        # Initializing
        def __init__(self):
                print('Employee created.')

      # Deleting (destructor)
      def __del__(self):
                print('Destructor called, Employee deleted.')

obj = Employee()        #Employee created.
del  obj                # Destructor called, Employee deleted.
```

```
class Point:
   def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
   def __del__(self):
        print (“Point destroyed")
```

```
pt1 = Point()
pt2 = pt1
pt3 = pt1
print (id(pt1), id(pt2), id(pt3)
# prints the ids of the obejcts)
del pt1
del pt2
del pt3
```

When the above code is executed, it produces the following result-

3083401324 3083401324 3083401324

Point destroyed

Inheritance in Python

- 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.
  – It is transitive in nature

```
class   test():                                  #Parent / Base  class
    def   fun1(self):
          print("test class function")

class   check(test):                             #Child / Derived  class
    def   fun2(self):
          print("check class function")
```

```
class    test():
    def   fun1(self):
            print("test class function")


class   check(test):
    def   fun2(self):
            print("check class function")
```

```
T = test()

C = check()


T.fun1()

C.fun1()
C.fun2()
```

```python
class Person(object):
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def isEmployee(self):
        return False

class Employee(Person):
    def isEmployee(self):
        return True
    def salary(self, sal):
        self.salary = sal
        return self.salary
```

```python
p1 = Person("Geek1")

# An Object of Person
print(p1.getName())
print(p1.isEmployee())

e1= Employee("Geek2")

# An Object of Employee
print(e1.getName())
print(e1.isEmployee())
print(e1.salary(1000))
```

Built-In Class Attributes

- Every Python class keeps the following built-in attributes and they can be accessed using dot operator like any other attribute −

  __dict__: Dictionary containing the class's namespace.

  __doc__: Class documentation string or none, if undefined.

  __module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode.

  __name__: Class name.

  __bases__: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Class Inheritance

- It refers to defining a new class with little or no modification to an existing class.
- The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

```
class BaseClass:
    Body of base class class

class DerivedClass(BaseClass):
    Body of derived class
```

```
class polygon:
    def __init__(self, sides):
        self.n=sides
        self.s=[int(input("Enter sides")) for i in range(self.n)]
    def display(self):
        for i in range(self.n):
            print(self.s[i])
```

```
a=polygon(3)
Enter side 12
Enter side 34
Enter side 45

a.display()
12
34
45
```

```
class  triangle(polygon):
        def __init__(self):
                polygon.__init__(self, 3)            #super().__init__(3)
        def area(self):
                a,b,c=self.s
                S=(a+b+c)/2
                ar=(S*(S-a)*(S-b)*(S-c))**0.5
                print(ar)

t=triangle()
```

- Enter side 11
- Enter side 223
- Enter side 122

```
t.area()
```

```python
class A:
    def fun(self):
        print("A")
class B(A):
    def fun(self):
        super().fun()
        print("B")
class C(A):
    def fun(self):
        super().fun()
        print("C")
```

```python
class D(B,C):
    def fun(self):
        super().fun()
        print("D")


d=D()
d.fun()
```

<u>OUTPUT:</u>

A

C

B

D

Method resolution order

- D.__mro__
- (__main__.D, __main__.B, __main__.C, __main__.A, object)

```
class Base(object):
    def __init__(self, x):
        self.x = x

class Derived(Base):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y
    def printXY(self):
        print(self.x, self.y)


d = Derived(10, 20)
d.printXY()                                  #10, 20
```

Data Hiding

- In Python, we use double underscore (Or __) before the attributes name and those attributes will not be directly visible outside.

```
class MyClass:
    __hiddenVariable = 0
    def add(self, increment):
        self.__hiddenVariable += increment
        print (self.__hiddenVariable)

myObject = MyClass()
myObject.add(2)
myObject.add(5)

# This line causes error
print (myObject.__hiddenVariable)
```

- We can access the value of hidden attribute by a tricky syntax:

```
class MyClass:
    # Hidden member of MyClass
    __hiddenVariable = 10

myObject = MyClass()
print(myObject._MyClass__hiddenVariable)
```

Printing Objects

- Printing objects gives us information about objects we are working with.
- In python this can be achieved by using __repr__ or __str__ methods.

```
class Test:
        def __init__(self, a, b):
              self.a = a
              self.b = b

        def __repr__(self):
              return  “({0},{1})”.format(self.a, self.b)

         def __str__(self):
              return “({0},{1})”.format(self.a, self.b)
```

```
t = Test(1234, 5678)
print(t)                    # This calls __str__()
print([t])                  # This calls __repr__()
```

- If no __str__ method is defined, print t (or print str(t)) uses __repr__.
- If no __repr__ method is defined then the default is used

Method Overriding in Python

- Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its super class or by one of its super classes.
- The implementation in the subclass overrides the implementation of the super class by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.

Method Overloading(does not work in PYTHON)

- Overloading is the ability to define the same method, with the same name but with a different number of arguments and types.
- It's the ability of one function to perform different tasks, depending on the number of parameters or the types of the parameters.
- If we need such a behavior, we can simulate it with default parameters.

Python Operator Overloading

- You can change the meaning of an operator in Python depending upon the operands used.
- This practice is known as operating overloading.

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

p1 = Point(2,3)
p2 = Point(-1,2)
print(p1)       <__main__.Point object at 0x00000000031F8CC0>
p1 + p2

Traceback (most recent call last): ...
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

```
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)

>>> p1 = Point(2,3)
>>> print(p1)
(2,3)
>>> str(p1)
'(2,3)'
>>> format(p1)
'(2,3)'
```

Overloading the + Operator in Python

```
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(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
(1,5)
```

Operator Overloading Special Functions in Python		
Operator	**Expression**	**Internally**
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)

Comparison Operator Overloading in Python		
Operator	**Expression**	**Internally**
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

Overloading Comparison Operators in Python

```
class Point:
        def __init__(self, x = 0, y = 0):
                self.x = x
                self.y = y
        def __lt__(self,other):
                smag = (self.x ** 2) + (self.y ** 2)
                omag = (other.x ** 2) + (other.y ** 2)
                return smag < omag
```

```
>>> Point(1,1) < Point(-2,-3)
True
>>> Point(1,1) < Point(0.5,-0.2)
False
>>> Point(1,1) < Point(1,1)
False
```

Composition

- In object-oriented programming, **delegation** refers to evaluating a member (property or method) of one object (the receiver) in the context of another, original object (the sender).
  - ***Explicit delegation*** **(composition)**
  - ***Implicit delegation*** **(inheritance).**
- Usually composition is said to be a very generic technique that needs no special syntax, while inheritance and its rules are strongly dependent on the language of choice.

```python
class test:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def show(self):
        print(self.x, self.y)
```

```python
class demo:
    def __init__(self , a, b, c):
        self.obj = test(a, b)
        self.c = c
    def show1(self):
        self.obj.show()
        print(self.c)

d = demo(1,2,3)
d.show1()
```

Output:

1 2

3

Class method

- The @classmethod decorator, is a built in function decorator that is an expression that gets evaluated after your function is defined.
- A class method receives the class as implicit first argument, just like an instance method receives the instance.
- A class method is a method which is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class.
- For example it can modify a class variable that will be applicable to all the instances.

Static Method

- A static method does not receive an implicit first argument.
  Syntax:

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
            ...
```

- **returns:** a static method for function fun.
- A static method is also a method which is bound to the class and not the object of the class.
- A static method can’t access or modify class state.

Class method vs Static Method

- A class method takes cls as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can't access or modify it.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

```
from datetime import date
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def fromBirthYear(cls, name, year):
        return   cls(name, date.today().year - year)
    @staticmethod
    def isAdult(age):
        return age > 18
```

```
person1 = Person('Maya', 21)
person2 = Person.fromBirthYear('May', 1996)

print (person1.age)
print (person2.age)

# print the result
print(Person.isAdult(22))
```

Output

- 21 22 True

Instance method/static method/class method

```
class A:
    def foo(self,  x):
        print ("executing foo(%s,%s)" %(self , x))

    @classmethod
    def class_foo(cls,x):
        print ("executing class_foo(%s,%s)" %(cls,  x))

    @staticmethod
    def static_foo(x):
        print ("executing static_foo(%s)"%(x))

a=A()
a.foo(1)
A.class_foo(11)
A.static_foo(111)
```

Viewing Class Dictionaries

- At the heart of all this is a dictionary1 that can be accessed by

vars(ClassName**)**