Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Object Oriented Programming (OOP) in Python

- OOP is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
- For instance,
  - an object could represent a person with
    - properties like a name, age, and address and
    - behaviors such as walking, talking, breathing, and running. Or
  - an object could represent an email with
    - properties like a recipient list, subject, and body and
    - behaviors like adding attachments and sending.

- OOP is an approach for

  - modeling concrete, real-world things, like Cars, Tiger, Dog, Table, Room, etc.. .....
  - modeling relations between things, like companies and employees, students and teachers, and so on.
- OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

A language has few types. It cannot provide all possible types we may want to have – like desks, projector, chalkpiece.

A language should provide a mechanism to make our own type. That is called a class. A class is a type and implementation. A variable of the class type is called an object.

How to Define a Class

All class definitions start with the class keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

In [1]:
```
class Dog:
    pass

print(Dog)          #<class '__main__.Dog'>
print(type(Dog))    #<class 'type'>
```

```
<class '__main__.Dog'>
```

```
<class 'type'>
```

The body of the Dog class consists of a single statement: the pass keyword. pass is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

The Dog class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Dog objects should have. There are a number of properties that we can choose from, including name, age, color, and breed. To keep things simple, we'll just use name and age.

In [5]:
```
class Dog:
    def __init__(self, name1, age1):
        self.name = name1     # creates an attributes called name and age and
        self.age = age1       #   assigns to it the value of the name and age parameters.


    def display(self):
        print('name =',self.name, ' age =',self.age)

# Creates an object named dog1 and
#  dog1 is initialized by calling Dog.__init__(dog1,'Ruby',1.5)
dog1=Dog('Ruby',1.5)

dog2=Dog('Sam',0.8)    # Dog.__init__(dog2,'Sam',0.8)
dog3=Dog('puppy',2)

dog1.display()
dog2.display()
dog3.display()
```

```
name = Ruby  age = 1.5
name = Sam  age = 0.8
name = puppy  age = 2
```

The properties that all Dog objects must have are defined in a method called **.__init__()**. Every time a new Dog object is created, **.__init__()** sets the initial state of the object by assigning the values of the object's properties. That is, **.__init__()** initializes each new instance of the class.

You can give **.__init__()** any number of parameters, but the first parameter will always be a variable called self. When a new class instance is created, the instance is automatically passed to the self parameter in **.__init__()** so that new attributes can be defined on the object.

Instance Attributes

Attributes (or data members) created in **.__init__()** are called instance attributes. An instance attribute's value is specific to a particular instance of the class. All Dog objects have a name and an age, but the values for the name and age attributes will vary depending on the Dog instance.

Note: Class in OOPs allows you to package data and functionality together by concept, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine.

A class can have its attributes and behaviour.

In [1]:
```
#a class can have functions(behaviour)

class Ex1:
    def foo():
        print("foo of Ex")

Ex1.foo()
```

```
foo of Ex
```

The function(behaviour) foo belongs to Ex1 and is invoked using the class name.

Class attributes

Class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`

For example, the following Dog class has a class attribute called species with the value "Canis familiaris":

In [6]:
```
class Dog:
    species = "Canis familiaris"  # Class attribute

    def __init__(self, name1, age1):
        self.name = name1          # instance attribute
        self.age = age1            # instance attribute

    def display(self):
        print('name =',self.name, ' age =',self.age)

print('Accessed using class name:',Dog.species)

dog1=Dog('Ruby',1.5)   # Dog.__init__(dog1,'Ruby',1.5)
dog2=Dog('Sam',0.8)    # Dog.__init__(dog2,'Sam',0.8)
dog3=Dog('puppy',2)    # Dog.__init__(dog3,'puppy',2)

print('Accessed using dog1 instance:',dog1.species)
print('Accessed using dog2 instance:',dog2.species)
print('Accessed using dog3 instance:',dog3.species)
#print('-----------------')
#Dog.species="Golden Retriever"
#print(dog1.species)
#print(dog2.species)
#print(dog2.species)
print('------------------')
dog1.species="Golden Retriever1"
dog2.species="Golden Retriever2"

print('dog1.species="Golden Retriever1"')
print('dog2.species="Golden Retriever2"')
print('Accessed using class name:',Dog.species)
print('Accessed using dog1 instance:',dog1.species)
print('Accessed using dog2 instance:',dog2.species)
print('Accessed using dog3 instance:',dog3.species)
print('------------------')
print('id(Dog.species) =',id(Dog.species))
print('id(dog1.species) =',id(dog1.species))
print('id(dog2.species) =',id(dog2.species))
print('id(dog3.species) =',id(dog3.species))
print('------------------')
dog1.display()
dog2.display()
dog3.display()
```

```
Accessed using class name: Canis familiaris
Accessed using dog1 instance: Canis familiaris
Accessed using dog2 instance: Canis familiaris
Accessed using dog3 instance: Canis familiaris
------------------
dog1.species="Golden Retriever1"
dog2.species="Golden Retriever2"
Accessed using class name: Canis familiaris
Accessed using dog1 instance: Golden Retriever1
Accessed using dog2 instance: Golden Retriever2
Accessed using dog3 instance: Canis familiaris
------------------
id(Dog.species) = 1718992474960
id(dog1.species) = 1718993015632
id(dog2.species) = 1718993016752
id(dog3.species) = 1718992474960
------------------
name = Ruby  age = 1.5
name = Sam  age = 0.8
name = puppy  age = 2
```

In [11]:

```python
class CSStudent:

    branch = 'CSE'    # Class Variable (or static variable)

    def __init__(self,name,roll):
        self.name = name            # Instance Variable
        self.roll = roll            # Instance Variable

# Objects of CSStudent class
s1 = CSStudent('Ram', 11)
s2 = CSStudent('Kishan', 21)

print('Branch =',s1.branch)
print('Name =',s1.name)
print('Roll No. =',s1.roll)
print()
print('Branch =',s2.branch)
print('Name =',s2.name)
print('Roll No. =',s2.roll)

# Class variables can be accessed using class name also
print('Branch name accessed using class name:',CSStudent.branch)
```

```
Branch = CSE
Name = Ram
Roll No. = 11

Branch = CSE
Name = Kishan
Roll No. = 21
Branch name accessed using class name: CSE
```

Use class attributes to define properties that should have the same value for every class instance.

The attribute species with the value "Canis familiaris" belongs to Dog and is accessed using the class name.

Use instance attributes for properties that vary from one instance to another.

In [5]:

```python
# a class can have attributes(fields or variables)
class Ex2:
    a = "test"

print(Ex2.a)
```

```
test
```

The attribute a with the value “test” belongs to Ex2 and is accessed using the class name.

Instantiate an Object in Python

Creating a new object from a class is called instantiating an object. You can instantiate a new Ex3 object by typing the name of the class, followed by opening and closing parentheses:

In [4]:
```
class Ex3:
    pass

a = Ex3()  # Instantiating an Object

print(a, type(a))
```

```
<__main__.Ex3 object at 0x0000015942981D30> <class '__main__.Ex3'>
```

This funny-looking string of letters and numbers is a memory address that indicates where the Ex3's object (i.e., a) is stored in your computer’s memory.

self in Python class

self represents the instance of the class. By using the “self” we can access the attributes of the class in python. It binds the attributes with the given arguments.

Self is always pointing to Current Object.

In [10]:
```
#it is clearly seen that self and obj is referring to the same object

class check:
    def __init__(self):
        print("Address of self = ",id(self))

obj = check()
print("Address of class object = ",id(obj))
```

```
Address of self =  1719012605472
Address of class object =  1719012605472
```

Self must be provided as a First parameter to the Instance method and constructor. If you don’t provide it, it will cause an error.

In [ ]:
```
# Self is always required as the first argument
class check:
    def __init__():
        print("This is Constructor")

object = check()
print("Worked fine")
```

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:

In [13]:
```
class Person:
  def __init__(myobject, name, age):
    myobject.name = name
    myobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)
```

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

```
Hello my name is Raj
```

Constructors in Python

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.

In Python the
`__init__`
method is called the constructor and is always called when an object is created.

In [13]:
```
class Ex4:
    def __init__(): # gives an error bcoz self parameter is missing
        print("constructor called")

a = Ex4()    ''' An object a of type Ex4 is created and calls Ex4.__init__(a)'''
```

```
constructor called
```

The object creation statement a = Ex4() conceptually becomes Ex4.**__init__**(a).

The class instance(i.e., object) is passed as the first argument to the constructor **__init__**.

Even though the argument(i.e., class instance or object) is implicit in Python implicit constructor call, we require an explicit appearance of object parameter in constructor. This can be given any name - is normally called self (like this parameter in java/C++).

In [9]:
```
class Ex4:
    def __init__(self):    # Constructor taking no arguments
        print("Constructor called")
        print('self : ', self)

# Constructor (__init__()) is called immediately upon object creation
a = Ex4()  # Ex4.__init__(a)
print('a : ', a)
```

```
Constructor called
self :  <__main__.Ex4 object at 0x0000023117751520>
a :  <__main__.Ex4 object at 0x0000023117751520>
```

Observe that the both the outputs have the same value indicating self and a refer to the same object.

Example: When we do not declare a constructor

In this example, we do not have a constructor but still we are able to create an object for the class.

In [10]:
```
class DemoClass:
    num = 101

    def read_number(self):
        print(self.num)

# creating object of the class
obj = DemoClass()
```

```
# calling the instance method using the object obj
obj.read_number()
```

```
101
```

Instance attributes

An object can have attributes. These are normally added and initialized in the constructor.

In [7]:
```
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
        # print(length) # error, all names should be fully qualified
        print("inside __init__(), the length and breadth are", self.length, self.breadth

# create a Rect object r1 and initialize with length 20 and breadth 10
r1 = Rect(20, 10)
print('Using object r1, the length and bradth are',r1.length, r1.breadth)
```

```
inside __init__(), the length and breadth are 20 10
Using object r1, the length and bradth are 20 10
```

Also, observe that these attributes can be accessed anywhere with a fully qualified name.

We can create any number of objects - we have to invoke the constructor to initialize them.

In [ ]:
```
r1 = Rect(20, 10)
r2 = Rect(40, 30)
r3 = Rect(60, 40)
```

In the below code, all instances data members are initialized with the same values 20 and 10.

In [19]:
```
class Rect:

    def __init__(self):      # Constructor taking no arguments
        self.length = 20     # All instances data members are initialized
        self.breadth = 10    #   with the same values 20 and 10.
        #print("inside __init__(), the length and bradth are", self.length, self.breadth

r1 = Rect()
print('In object r1, the length and bradth are',r1.length, r1.breadth)
r2 = Rect()
print('In object r2, the length and bradth are',r2.length, r2.breadth)
```

```
In object r1, the length and bradth are 20 10
In object r2, the length and bradth are 20 10
```

In [6]:
```
print(r1==r2)
```

```
False
```

When you compare r1 and r2 using the == operator, the result is False. Even though r1 and r2 are both instances of the Rect class, they represent two distinct objects in memory.

Note:

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect.

Although the attributes are guaranteed to exist, their values can be changed dynamically:

In [22]:
```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        print(f"{self.name} is {self.age} years old")

buddy = Dog("Buddy", 9)
miles = Dog("Miles", 4)

buddy.description()
miles.description()

buddy.age = 10
buddy.description()

print('id(buddy.species) == id(miles.species) is ',id(buddy.species) == id(miles.species
print('buddy.species =',buddy.species)
print('miles.species =',miles.species)

print('After assignment statement: miles.species = "Felis silvestris"')
miles.species = "Felis silvestris"
print('id(buddy.species) == id(miles.species) is ',id(buddy.species) == id(miles.species
print('buddy.species =',buddy.species)
print('miles.species =',miles.species)
```

```
Buddy is 9 years old
Miles is 4 years old
Buddy is 10 years old
id(buddy.species) == id(miles.species) is  True
buddy.species = Canis familiaris
miles.species = Canis familiaris
After assignment statement: miles.species = "Felis silvestris"
id(buddy.species) == id(miles.species) is  False
buddy.species = Canis familiaris
miles.species = Felis silvestris
```

In this example, you change the .age attribute of the buddy object to 10. Then you change the .species attribute of the miles object to "Felis silvestris", which is a species of cat. That makes Miles a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

There is no constructor overloading in Python.

In [13]:
```
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b
        print('First Constructor is used')

    def __init__(self, l, b):
        self.length = l
        self.breadth = b
        print('Second Constructor is used')

r1 = Rect(10,20)
print('l =',r1.length)
print('b =',r1.breadth)
```

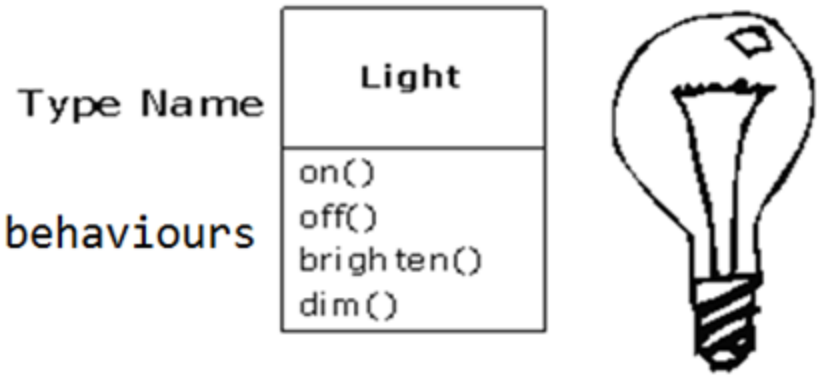
```
Second Constructor is used
```

```
l = 10
b = 20
```

What this means is Python rebinds the name **init** to the new method. This means the first declaration of this method is inaccessible now.

Instance Methods

Our class can also have behaviour through functions in the class.



In the below code, an object of class Rect contains length and breadth. Given the rectangle, we can extract the length and the breadth to find the area. So, the function area(instance method) is invoked with an object and does not require any other parameter.

The object encapsulates both attributes and behaviour. Encapsulation is putting together data and functions (attributes and behaviour).

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like .**init**(), an instance method's first parameter is always self.

In [7]:
```
class Rect:
    def __init__(self, l, b):
        self.length = l
        self.breadth = b

    # instance method
    def area(self):
        return self.length * self.breadth

# initialize a Rect object r1 with length 20 and breadth 10
r1 = Rect(20, 10)  #creates an object r1 and calls Rect.__init__(r1,20,10)
r2 = Rect(40, 30)
print('area of r1 : ', r1.area())    # Rect.area(r1)
print('area of r2 : ', r2.area())    # Rect.area(r2)
```

```
area of r1 :  200
area of r2 :  200
```

Let us look at a few more behaviours.

The change_length method requires the new length apart from the object attributes length and breadth. We require one explicit argument while calling change_length method and two explicit parameters in defining change_length method. The first parameter always refers to the object through which the call is made.

In [10]:
```
# Rect.change_length, Rect.change_breadth
class Rect:
```

```
    def __init__(self, l, b):
        self.length = l
        self.breadth = b

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

    def change_length(self, l):
        self.length = l

    def change_breadth(self, b):
        self.breadth = b

    def disp(self):
        print('length : ', self.length)
        print('breadth : ', self.breadth)

r1 = Rect(20, 10)        #Rect.__init__(r1,20,10)

print("before change length ")
r1.disp()                #Rect.disp(r1)

r1.change_length(40)     #Rect.change_length(r1,40)
print("after change length ")
r1.disp()

print("before change breadth ")
r1.disp()

r1.change_breadth(30)
print("after change breadth ")
r1.disp()
```

```
before change length
length :  20
breadth :  10
after change length
length :  40
breadth :  10
before change breadth
length :  40
breadth :  10
after change breadth
length :  40
breadth :  30
```

Inheritance in Python

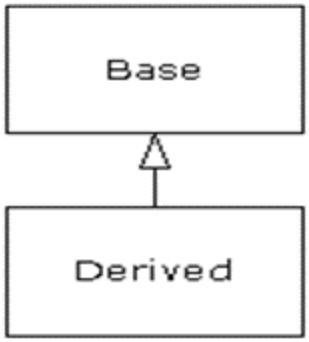
Inheritance is a mechanism wherein a new class is derived from an existing class. In Python, classes may inherit or acquire the attributes (data) and behaviors (functions) of other classes.

A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass.

Inheritance is the process wherein characteristics are inherited from ancestors. Similarly, in Python, a subclass inherits the attributes and behaviors of its superclass (ancestor).

For example, a vehicle is a superclass and a car is a subclass. The car (subclass) inherits all of the vehicle's properties. The inheritance mechanism is very useful in code reuse.

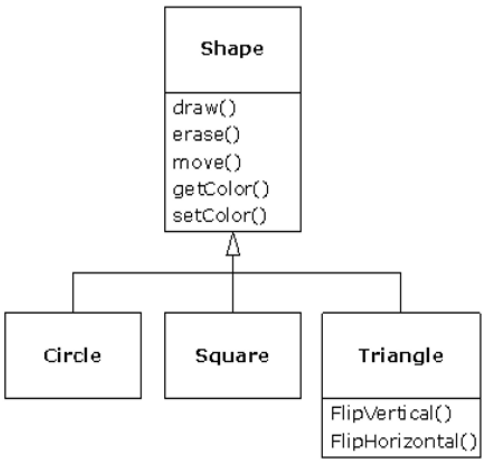
Inheritance is a mechanism to capture the commonality between classes. This is used to create class or type hierarchy. This indicates the relation of “ISA” between classes.



A base type contains all of the attributes and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It’s nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with inheritance.

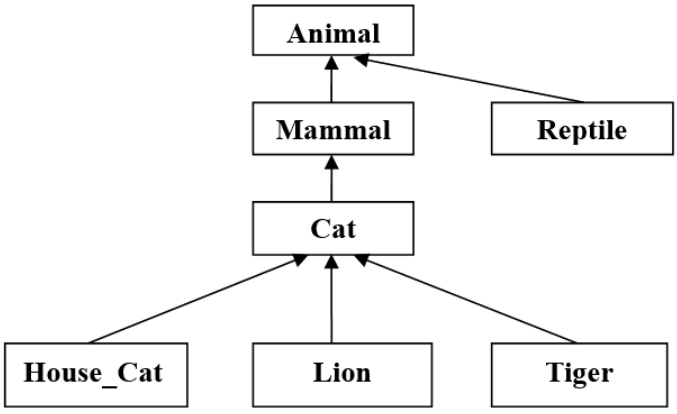
Example 1:



The base type is “shape,” and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc.

From this, specific types of shapes are derived (inherited)—circle, square, triangle, and so on—each of which may have additional characteristics and behaviors.

Example 2:



Now, based on the above example, In Object Oriented terms, the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Cat is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Cat IS-A Mammal
- Hence : Cat IS-A Animal as well

A cat is a mammal. Tiger is a cat. House_cat is a cat. Lion is a cat. All these have common characteristics. We can address all of them as cats. They walk on their toes. They climb trees. There could be differences in the way they walk or they climb. Because of common characteristics, we can consider a tiger or a lion as a cat. This helps in maintenance of programs.

We call this “is a” relationship between classes as inheritance.

Example 3:

In [4]:
```
class P2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        print(self)

    def dispay(self):
        print ("x : ", self.x)
        print ("y : ", self.y)


class P3D(P2D):       # P3D inherits from P2D. An object of P3D gets everything P2D has.
    def __init__(self, x, y, z):
        P2D.__init__(self, x, y)
        #self.x = x
        #self.y = y
        self.z = z
        print(self)

p3 = P3D(11, 12, 13)  #  P3D.__init__(p3,11,12,13)
p3.dispay()      # no function in the derived class; calls the base class function by def

print(isinstance(p3, P3D))
print(isinstance(p3, P2D))
```

```
<__main__.P3D object at 0x00000221C0777250>
x :  11
y :  12
True
True
```

A point in three dimensions(class P3D) is also a point in two dimensions(P2D).

class P3D(P2D):

The above statement indicates that relationship. We say that P3D is a derived class and P2D is the base class.

An object of derived class with a reference to an embedded object of base class – normally referred to as the base class sub object. When we create an object of the derived class, we invoke the constructor of the derived class which in turn calls the base class constructor on the base class sub object.

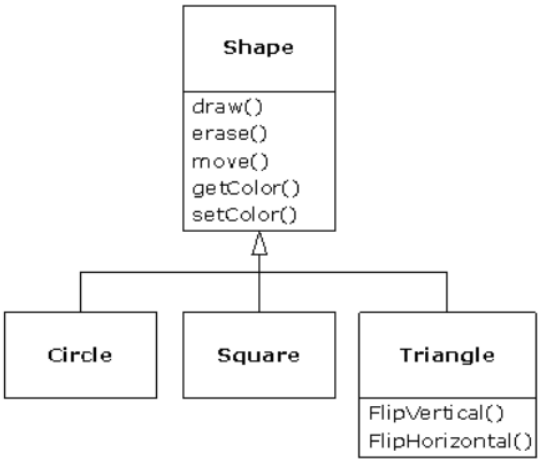
The base class provides a default implementation of the behaviour for the derived class.

p3.disp() # no function in the derived class; calls the base class function by default

As there is no function called disp in the derived class, the base class function gets called.

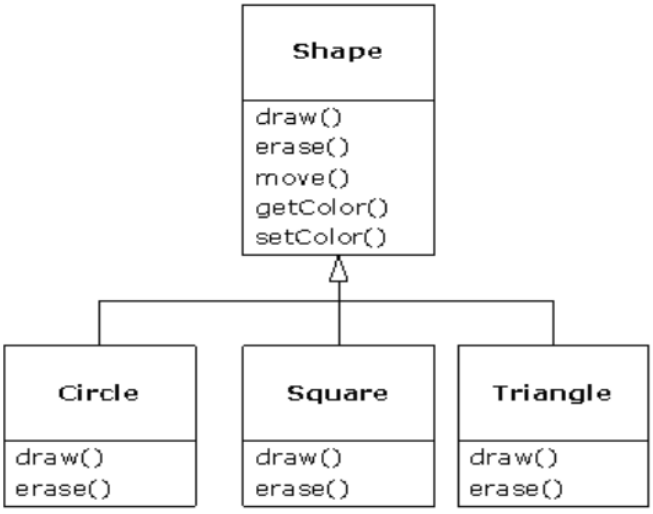
You have two ways to differentiate your new derived class from the original base class.

1) The first is quite straightforward: You simply add brand new functions to the derived class. These new functions are not part of the base-class interface.



2) Overriding:

The second and more important way to differentiate your new class is to change the behavior of an existing base-class function. This is referred to as overriding that function.



To override a function, you simply create a new definition for the function in the derived class. You're saying, " I'm using the same function interface(or signature) here, but I want it to do something different for my new type."

Example:

In [9]:
```
class P2D:
```

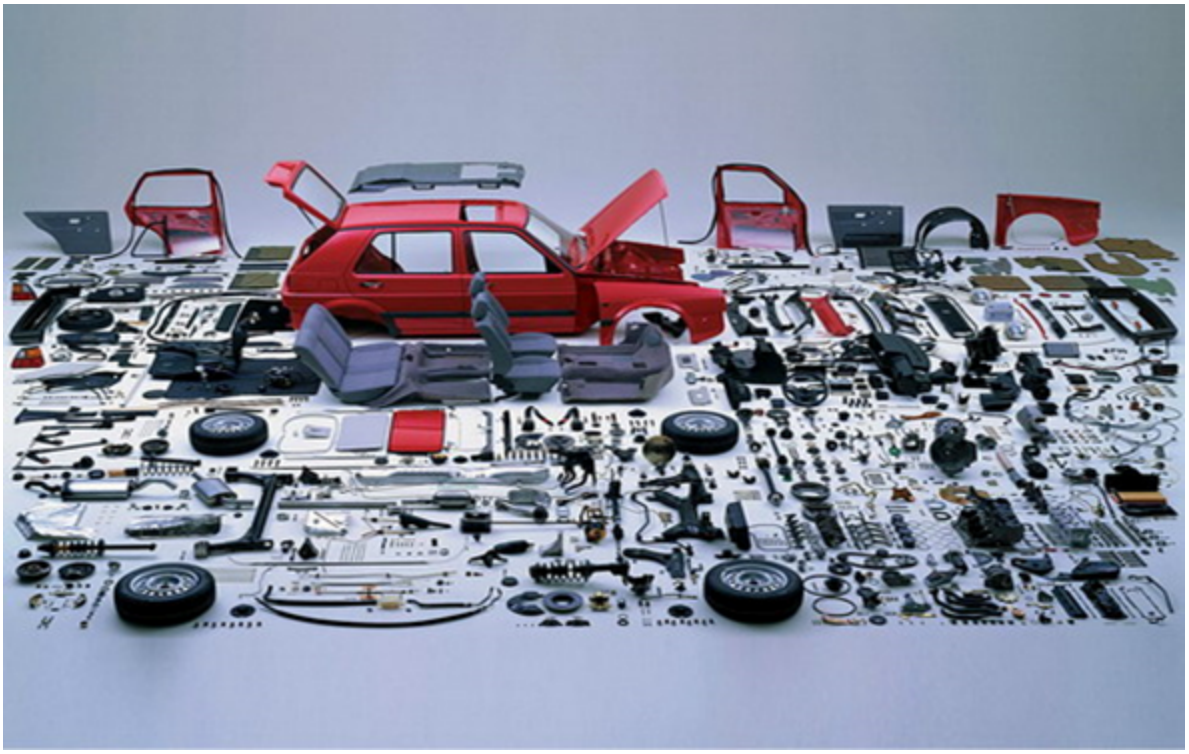
```
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def display(self):
        print ("x : ", self.x)
        print ("y : ", self.y)

class P3D(P2D) :
    def __init__(self, x, y, z):
        P2D.__init__(self, x, y)
        self.z = z

    #overriding the base class function
    def display(self):
        P2D.display(self) # delegate work to the base class
        print("z : ", self.z)

p3 = P3D(11, 12, 13)
p3.display()
```

```
x :  11
y :  12
z :  13
```

It is possible for the derived class to provide its own function disp. This concept is called overriding. The derived class modifies the function of the base class. In many such cases, this overriding function may in turn call the base class function.

Composition:

Composing a new class from existing classes is called composition. Composition is often referred to as a “has-a” relationship, as in “A car has an engine.”



Example 1:

Your Cat class can be composed by another object of type Tail. Composition allows you to express that relationship by saying a Cat has a Tail, Paws, Whiskers, Claws, Teeth, … .

Example 2:

In the below code, MyEvent object has MyDate object as part of it. An Event occurs on a date. An event is not a date. A date is not an event. An event has a date as part of it. So observe that the constructor of MyEvent class initializes a field called date by creating and initializing an object of MyDate class.

All functions of the MyEvent class in turn invoke the functions of MyDate class through the attribute date – this is called delegation or forwarding.

We have been using the function str on different types like int. When we invoke this function, a special function **str** of that class will be called. We can also provide such a function in our class to convert an object of our class to a string. In the print context, this function gets called automatically.

In [13]:
```
class MyDate:
    def __init__(self, dd, mm, yy):
        self.dd = dd
        self.mm = mm
        self.yy = yy

    def __str__(self):
        return str(self.dd) + "/" + str(self.mm) + "/" + str(self.yy)

    def key(self):
        return self.yy * 365 + self.mm * 30 + self.dd

d = MyDate(15, 8, 1947)
print(d) # MyDate.__str__(d)
print(d.key())
```

```
15/8/1947
710910
```

In [14]:
```
class MyEvent:
    def __init__(self, dd, mm, yy, detail):
        self.date = MyDate(dd, mm, yy)
        self.detail = detail

    def __str__(self):
        return str(self.date) + " => " + self.detail

    def key(self):
        #return self.detail
        return self.date.key()  # return MyDate.key(self.date)

e = MyEvent(15, 8, 1947, "Independence Day")
print(e)

print(e.key()) # print(MyEvent.key(e))

print(MyEvent.key(MyEvent(15, 8, 1947, "Independence Day")))
```

```
15/8/1947 => Independence Day
710910
710910
```

The rest of the code creates a list of user defined objects – creates a list of events. Then the list is sorted and

displayed using for statement.

We would like sort these dates based on the date. We provide keyword parameter key which converts the date into a single number.

The callback MyEvent.key in turn delegates the work to MyDate.key. This function converts a given date to a single integer by using some approximate formula to count the number of days from the beginning of the era.

The sorted function converts each event into this number and sorts the events based on this number.

In [10]:
```
mylist = [
            MyEvent(26, 1, 1956, "Republic Day"),
            MyEvent(1, 11, 1973, "Karnataka born"),
            MyEvent(2, 10, 1868, "Gandhi jayanthi"),
            MyEvent(15, 8, 1947, "Independence Day"),
            MyEvent(16, 12, 1971, "Amar sonar bangla")
         ]

for e in sorted(mylist, key = MyEvent.key):
    print(e)
```

```
2-10-1868 => Gandhi jayanthi
15-8-1947 => Independence Day
26-1-1956 => Republic Day
16-12-1971 => Amar sonar bangla
1-11-1973 => Karnataka born
```

References:

1. oo.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://docs.python.org/
3. https://realpython.com/instance-class-and-static-methods-demystified/
4. https://realpython.com/python3-object-oriented-programming/