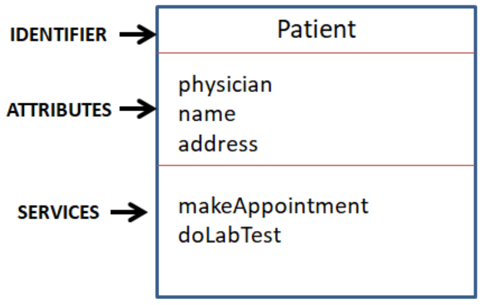
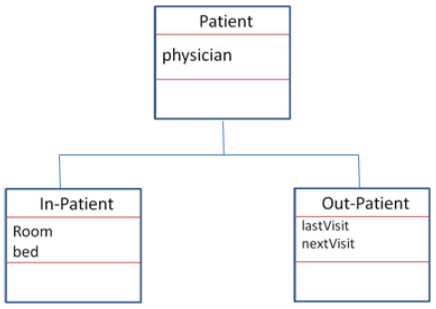
Class and Objects by Swapnil Saurav

OBJECT ORIENTED PROGRAMMING IN PYTHON

Classes and objects are the two main aspects of object oriented programming. A class creates a new type where objects are instances of the class. An object is defined as a real world entity related to the problem domain, with "crisply defined boundaries". Objects are encapsulated with attributes (called as fields in Python) and behaviour or services (called as methods in Python).





Patients are either in-patients or out-patients. The physician attribute of patients is inherited by both in-patients and out-patients.

OOP CONCEPTS IN PYTHON

Python is an object-oriented programming language. We can easily create and use classes and objects in Python.

Major object-oriented concepts of Python programming language are given below –

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Object
Object is an entity that has state and behaviour. It may be physical and logical. For example: mouse, keyboard, chair, table, pen etc. Everything in Python is an object, and almost everything has attributes and methods.

Class
Class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have a student class then it should contain an attribute and method i.e. an email id, name, age, roll number etc.

Method
Method is a function that is associated with an object. In Python, method is not unique to class instances. Any object type can have methods.
Inheritance
Inheritance specifies that one object acquires all the properties and behaviors of parent object. By using inheritance we can define a new class with a little or no changes to the existing class. The new class is

known as derived class or child class and from which it inherits the properties is called base class or parent class. It provides re-usability of the code.

Polymorphism
Polymorphism defines that one task can be performed in different ways. For example: We have a class animal and all animals talk. But they talk differently. Here, the "talk" behaviour totally depends on the animal. So, the abstract "animal" does not actually "talk", but specific animals have a concrete implementation of the action "talk".

Encapsulation
Encapsulation is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction
Data abstraction and encapsulation are synonymous as data abstraction is achieved through encapsulation.
Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the basic idea of what a function or a whole program does.

CLASS & OBJECTS IN PYTHON

Objects can store data using ordinary variables that belong to the object. Variables that belong to an object or class are referred to as fields. Objects can also have functionality by using functions that belong to a class. Such functions are called methods of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. A class is created using the class keyword. The fields and methods of the class are listed in an indented block.

```python
#Declaring Class
class Person:
    def say_hi(self):
        print("How are you?")

#Now initialize an object
p = Person()
#call method of the class
p.say_hi()
#We can the method in one line itself:
Person().say_hi()  #Initialize the class and call the method
```

Output:

```
Hello, how are you?
Hello, how are you?
```

Notice that the say_hi method takes no parameters but still has the self in the function definition.
The init method
There are many method names which have special significance in Python classes. The __init__ (double underscore init double underscore) method 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. Notice the double underscores is at both the beginning and at the end of the name.

Class and Object Variables

The data part, i.e. fields, are nothing but ordinary variables that are bound to the namespaces of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called name spaces. There are two types of fields - class variables and object variables which are classified depending on whether the class or the object owns the variables respectively. Class variables are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.
Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance. An example will make this easy to understand.
This example, you will see:
@classmethod: The classmethod() is an inbuilt function in Python, which returns a class method for a given function. classmethod() methods are bound to a class rather than an object. Class methods can be called by both class and object. These methods can be called with a class or with an object.

```
#Declaring Class Example
class Book:
    """Represents a Book class example"""
    #Below declaring a class variable counting total book count
    book_count = 0
    #Now Initialize the variable using init
    def __init__(self,title):  #Note self is added as one of the 
variable.
                         # The self keyword is used to represent an 
instance (object) of the given class.
        ''' Initialize the data'''
        self.title = title
        print("Initializing the title of the book: ", self.title)
        #When a book is created, it should increase the total book 
count by 1
        Book.book_count +=1  #Since its a class level variable, 
Class name.variable is used

    def remove(self):
        '''Removing book from the list'''
        print("{} is being removed from the 
list!".format(self.title))
        Book.book_count -= 1
        if Book.book_count <= 0:
            print("This was the last book in the shelf. You do not 
have any more book!")
        else:
            print("There are still {} books in the 
shelf!".format(Book.book_count))

    def say_hi(self):
        """Hello from the book class"""
        print("Hello from Class Book, I am being called by 
",self.title)

    @classmethod
    def how_many(cls):
        """Prints current number of books available"""
        print("Currently there are {} books in the
shelf.".format(cls.book_count))

####  Below code is used to create object of the class Book
#Creating objects and initializing the title using __init__ called
while creating objects
book1 = Book("The A to Z of Retail Management")
book2 = Book("Learn and Practice Python Programming")
book3 = Book("Boost your career")
#calling class variable using classname
print("Total number of books available in the shelf:
",Book.book_count)
#Calling functions
book1.say_hi()
book2.remove()
#calling class variable using object
print("Total number of books available in the shelf:
",book3.book_count)
```

Output:

```
Initializing the title of the book:  The A to Z of Retail Management

Initializing the title of the book:  Learn and Practice Python Programming

Initializing the title of the book:  Boost your career

Total number of books available in the shelf:  3

Hello from Class Book, I am being called by  The A to Z of Retail Management

Learn and Practice Python Programming is being removed from the list!

There are still 2 books in the shelf!

Total number of books available in the shelf:  2
```

Here, book_count belongs to the Book class and hence is a class variable. The title variable belongs to the object (it is assigned using self) and hence is an object variable. Thus, we refer to the population class variable as Book.book_count and not as self.book_count. We refer to the object variable name using self.title notation in the methods of that object. Instead of Book.book_count , we could have also used self.__class__.population because every object refers to it's class via the self.__class__ attribute.

init method is used to initialize the Book instance with a name and we increase the book_count by 1 since a book is added. In the remove method, we simply decrease the Book.book_count by 1. All class members are public. One exception: If you use data members with names using the double underscore prefix such as __privatevar , Python uses namemangling to effectively make it a private variable. Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects. This is only a convention and is not enforced by Python (except for the double underscore prefix).

Instance Attributes and Methods

An function defined in a class is called a "method". Methods have access to all the data contained on the instance of the object; they can access and modify anything previously set on `self`. Because they

use self, they require an instance of the class in order to be used. For this reason, they're often referred to as "instance methods".
If there are "instance methods", then surely there are other types of methods as well, right? Yes, there are, but these methods are a bit more esoteric. We'll cover them briefly here, but feel free to research these topics in more depth.

Static Methods
CLASS ATTRIBUTES are attributes that are set at the CLASS-LEVEL, as opposed to the INSTANCE-LEVEL. Normal attributes are introduced in the __init__ method, but some attributes of a class hold for ALL instances in all cases. For example, consider the following definition of a Car object:

```
class Car(object):
    wheels = 4
    def __init__(self, make, model):
        self.make = make
        self.model = model

##### Class objects
mustang = Car('Ford', 'Mustang')
print(mustang.wheels)
# 4
print(Car.wheels)
# 4
```

A Car always has four wheels, regardless of the make or model. Instance methods can access these attributes in the same way they access regular attributes: through self (i.e. self.wheels). There is a class of methods, though, called STATIC METHODS, that don't have access to self. Just like class attributes, they are methods that work without requiring an instance to be present. Since instances are always referenced through self, static methods have no self parameter.
The following would be a valid static method on the Car class:

```
class Car(object):
    ...
    def make_car_sound():
        print('VRooooommmm!')
```

No matter what kind of car we have, it always makes the same sound, we as per the above program. To make it clear that this method should not receive the instance as the first parameter (i.e. self on "normal" methods), the@staticmethod decorator is used, turning our definition into:

```
class Car(object):
    ...
    @staticmethod
    def make_car_sound():
        print('VRooooommmm!')
```

Class Methods

A variant of the static method is the class method. Instead of receiving the *instance* as the first *parameter*, it is passed the *class*. It, too, is defined using a decorator:

```
class Vehicle(object):
    ...
    @classmethod
    def is_motorcycle(cls):
        return cls.wheels == 2
```

INHERITANCE

One of the major benefits of object oriented programming is reuse of code and one of the ways this is achieved is through the inheritance mechanism. Inheritance is where one class (child class) inherits the members of another class (parent class). The benefit of inheritance is that the child class doesn't have to redeclare and redefine all the members which it inherits from the parent class. It is therefore a way to re-use code. Inheritance also enables you to write generic code which applies to all classes in an inheritance hierarchy. You can do this because you know that all child classes must contain the same members as the parent class.

While Object-oriented Programming is useful as a modeling tool, it truly gains power when the concept of inheritance is introduced. Inherticance is the process by which a "child" class derives the data and behavior of a "parent" class. Let's understand this with an example.

```
class School:
    """ Represents School """
    def __init__(self, name,age):
        self.name = name
        self.age = age
        print("Initialization of {} done for School
membership".format(self.name))

    def tell(self):
        """Tell my details when called"""
        print("Thanks for calling me. My name is {} and my age is
{}".format(self.name, self.age))

#Lets declare another class Teacher and since Teacher can use Name
and Age, we will make Teacher inherit
#the properties from School. Apart from the School property,
Teacher can also have its own properties
#which makes it different from School.


class Teacher(School):  #Inheritance done
    """Represents the Teachers of the school"""
    def __init__(self,name,age,salary):
        School.__init__(self, name,age) #Calling Parent's (School
class) to inherit values for Name and Age
        self.salary = salary
        print("Initialization of Teacher object is done.")

    def tell(self):
        School.tell(self) #Calling tell since it has inherited it
        print("My salary is: ",self.salary)

#Another class which will inherit School properties
class Student(School):
    """Represents a Student object"""
    def __init__(self, name, age, marks):
        School.__init__(self, name,age)
        self.marks = marks
        print("Initialization of Student object is done.")

    def tell(self):
        School.tell(self)  # Calling tell since it has inherited it
        print("The marks I scored is: ", self.marks)

###  Now let's create objects and call them
teacher1 = Teacher("Rahul", 45, 85000)
student1 = Student("Ashwin", 14, 630)
school1 = School("Sachin", 50)
#Create a list to store the objects
list = [teacher1, student1, school1]
for member in list:
    member.tell()
    #tell() works for all 3 objects
```

Output:

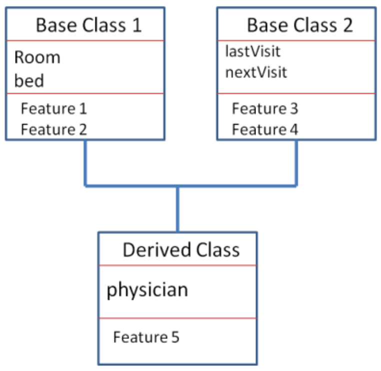
```
Initialization of Rahul done for School membership
Initialization of Teacher object is done.
Initialization of Ashwin done for School membership
Initialization of Student object is done.
Initialization of Sachin done for School membership
Thanks for calling me. My name is Rahul and my age is 45
My salary is:  85000
Thanks for calling me. My name is Ashwin and my age is 14
The marks I scored is:  630
Thanks for calling me. My name is Sachin and my age is 50
```

One of the most important rules of programming (in general) is "Don't Repeat Your Code", so we used Inheritance concept here. To use inheritance, we specify the base class names in a tuple following the class name in the class definition. Next, we observe that the init method of the base class is explicitly called using the self variable so that we can initialize the base class part of the object. This is very important to remember - Python does not automatically call the constructor of the base class, you have to explicitly call it yourself. We also observe that we can call methods of the base class by prefixing the class name to the method call and then pass in the self variable along with any arguments.
We can treat instances of Teacher or Student as just instances of the School when we use the tell method of the School class. Remember that the tell method of the subtype (child class) is called and not the tell method of the School (Parent) class. One way to understand this is that Python always starts looking for methods in the actual type, which in this case it does. If it could not find the method, it starts looking at the methods belonging to its base classes one by one in the order they are specified in the tuple in the class definition.

MULTIPLE INHERITANCE AND MULTILEVEL INHERITANCE

Multiple Inheritance in Python

A class can be derived from more than one base classes in Python. This is called multiple inheritance. In multiple inheritance, the features of all the base classes are inherited into the derived class (see the figure below).



The syntax for multiple inheritance is similar to single inheritance:

```
class Base1():
    ...
class Base2():
    ...

class Derived(Base1, Base2):
    ...
```

Multilevel Inheritance in Python

We also have Multilevel Inheritance. We can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python. In our example, Cube 'is a' Rectangle which 'is a' Shape.

```
class Shape():
    ...
class Rectangle(Shape):
    ...

class Cube(Rectangle):
    ...
```

Please refer below example to practice the above concepts:

```
class Point:
    ''' The class Point represents a 3D point
       Instance attributes: x,y,z '''
    def __init__(self):
        '''Default object initialization'''
        self.X = 0.0
        self.Y = 0.0
        self.Z = 0.0

    def __init__(self, x=0, y=0, z=0):
        '''Custom initialization'''
        self.X = x
        self.Y = y
        self.Z = z

    def __str__(self):
        '''Print out the String of this object's values'''
        return '({}, {}, {})'.format(
            str(self.X), str(self.Y), str(self.Z))

    def SetPoint(self, x=None, y=None, z=None):
        '''Set new locations for the point'''
        if x is not None:
            self.X = x
        if y is not None:
            self.Y = y
        if z is not None:
            self.Z = z

    def Distance(self, other):
        '''Returns the distance'''
        d = (((other.X - self.X) ** 2)
             + ((other.Y - self.Y) ** 2)
             + ((other.Z - self.Z) ** 2)) ** 0.5
        return d

class Shape:
    ''' A general Shape class '''
    def __init__(self, p, n):
        '''Set the name and vertices of the shape'''
        self.name = n
        self.vertices = p

    def Area(self):
        '''Caculate the area of the shapes'''
        raise NotImplementedError("Please implement the Area method!")
        # This containts a check to make sure subclasses implement this.
        return None

    def __str__(self):
        '''Definition for the print statement'''
        return "Shape: '{}' of type ({}) has {} points.".format(
            self.name, str(self.__class__.__name__),
            str(len(self.vertices)))

class Shape3D:
    ''' A general 3D Shape class '''
    def Volume(self):
        '''Caculate the Volume of the shapes'''
        raise NotImplementedError("Please implement the Volume method!")
        # This containts a check to make sure subclasses implement this.
        return None

class Rectangle(Shape):
    '''Rectangle is a shape with 4 vertices'''
    def __init__(self, p, n):
        '''Rectangle vs: lower left, upper left, upper right, lower
right'''
        if len(p) != 4:
            print('FATAL ERROR: A rectangle has 4 points.')
            return
        Shape.__init__(self, p, n)

    def Area(self):
        '''Area of a rectangle is length*width'''
        length = self.vertices[0].Distance(self.vertices[1])
        width = self.vertices[1].Distance(self.vertices[2])
        return length * width

class Circle(Shape):
    '''Circle is a shape with 1 vertex and a radius'''
    def __init__(self, p, r, n):
        '''Set the center point and radius '''
        self.radius = r
        if len(p) != 1:
            print('FATAL ERROR: A rectangle has 1 center point.')
            return
        Shape.__init__(self, p, n)

    def Area(self):
        '''Radius is pi*r^2 '''
        return 3.14159 * self.radius ** 2

    def Circumference(self):
        '''Caculate the Circle Circumference'''
        return self.radius * 2 * 3.14

    def __str__(self):
        '''the print statement: uses super print + radius info'''
        return Shape.__str__(self) + \
               " It has a radius of: {}.".format(str(self.radius))

class Cube(Rectangle, Shape3D):
    '''Cube is a 3d shape with 8 vertices'''
    def __init__(self, p, n):
        '''Rectangle vs: lower left, upper left, upper right, lower
right'''
        if len(p) != 8:
            print('FATAL ERROR: A Cube has 8 points.')
            return
        Shape.__init__(self, p, n)

    def Area(self):
        '''Area of a cube is length*width*height'''

        # this should be defined by lower left vertex - clockwise
        # then the upper left vertex - clockwise
        # hence the 5th vertex (id[4]) should be above the 1st (id[0])
        length = self.vertices[0].Distance(self.vertices[1])
        width = self.vertices[1].Distance(self.vertices[2])
        height = self.vertices[0].Distance(self.vertices[4])
        return length * width * height

    def Volume(self):
        '''Volume of a cube is a*3'''
        return self.Area() * 3

class Sphere(Circle, Shape3D):
    '''Sphere is a shape with 1 vertex and a radius'''
    def __init__(self, p, r, n):
        '''Set the center point and radius '''
        Circle.__init__(self, p, r, n)

    def Area(self):
        '''Radius is pi*r^2 '''
        return 4 * 3.14 * self.radius ** 2

    def Volume(self):
        '''Caculate the Sphere Circumference'''
        return 4 / 3 * 3.14 * self.radius * 3

# Remember this method gets executed sinces its our 'main'
def main():
    # Make a rectangle
    p1 = [Point(0.0, 0.0, 0.0), Point(0.0, 4.0, 0.0),
          Point(2.0, 3.0, 0.0), Point(3.0, 0.0, 0.0)]
    r = Rectangle(p1, "Rectangle 1")
    print(r)
    print("Rectangle area is: {:0.2f}".format(r.Area()))

    # make a Circle
    p2 = [Point(0.0, 0.0, 0.0)]
    c = Circle(p2, 4.0, "Circle 1")
    print(c)
    print("Circle area is: {:0.2f}".format(c.Area()))

    # Make a Cube
    p3 = [Point(0.0, 0.0, 0.0), Point(0.0, 3.0, 0.0),
          Point(2.0, 2.0, 0.0), Point(2.0, 0.0, 0.0),
          Point(0.0, 0.0, 4.0), Point(0.0, 3.0, 4.0),
          Point(2.0, 3.0, 3.0), Point(2.0, 1.0, 4.0)]
    cub = Cube(p3, "Cube 1")
    print(r)
    print("Cube area is: {:0.2f}".format(cub.Area()))
    print("Cube volume is: {:0.2f}".format(cub.Volume()))

    # Make a Sphere
    p4 = [Point(0.0, 1.0, 0.0)]
    s = Sphere(p4, 3.0, "Sphere 1")
    print(s)
    print("Sphere area is: {:0.2f}".format(s.Area()))
    print("Sphere volume is: {:0.2f}".format(s.Volume()))

if __name__ == "__main__":
    main()
```

ENCAPSULATION IN PYTHON

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those type of variables are known as private variable.

Abstraction is a mechanism which represent the essential features without including implementation details. A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

- Encapsulation: — Information hiding.
- Abstraction: — Implementation hiding.

Protected members: Protected member is (in C++ and Java) accessible only from within the class and it's subclasses. In Python we prefix the name of the member with a single underscore, by this we are telling others "don't touch this, unless you're a subclass". But there is no enforecement of any type.

Private members: Private members can be declared by adding "__" (double underscore) in front of the variable and function name can hide them when accessing them from out of class.

```
# Encapsulation: Python program to
# demonstrate protected and private members

# Creating a base class
class Base:
    def __init__(self):

        self.title = "Learn and Practice Python"
        # Protected member
        self._ed = "Edition 2"
        # Private member
        self.__author = "Swapnil Saurav"

# Creating a derived class
class Derived(Base):
    def __init__(self):
        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling public member of base class: ")
        print("Title of the book is ",self.title)   #Works fine
        print("Calling protected member of base class: ")
        print("Edition: ",self._ed)   #Works fine
        print("Calling private member of base class: ")
        # This throws error-private member cant be called
        #print("Author: ", self.__author)

#Creating objects
obj1 = Derived()
obj2 = Base()

# Calling protected member
print("Base class Public member: ", obj1.title)
print("Base class Private member: ", obj1._ed)
# Below will throw error-private member cant be called
#print("Base class Protected member: ", obj1.__author)
print("Derived class Public member: ", obj2.title)
print("Derived class Private member: ", obj2._ed)
# Below will throw error-private member cant be called
#print("Derived class Protected member: ", obj2.__author)
```

In the above example, only private members throw error, private members doesnt throw error because, Python doesn't have real private methods, so putting one underline (_) in the beginning of a method or attribute means you shouldn't access this method.But this is just convention. As you can see from the above example, we can still access the the variables with single underscore.

POLYMORPHISM

Car Deaslership Example

Let’s understand the concepts with a card dealership example. Imagine we run a car dealership. We sell all types of vehicles, from motorcycles to trucks. We set ourselves apart from the competition by our prices. Specifically, how we determine the price of a vehicle on our lot: Rs 1,00,000 x number of wheels a vehicle has. We also buy back our vehicles but at a discounted price based on the type of vehicle and the distance covered. We offer a flat rate - 10% of the km driven on the vehicle. For trucks, that rate is Rs 180,000, for cars, Rs 140,000 and for motorcycles, its Rs 80,000.

If we wanted to create a sales system for our dealership using Object-oriented techniques, how would we do so? We might have a`Sale` class, a `Customer` class, an `Inventory` class, and so forth, but we'd almost certainly have a `Car`, `Truck`, and `Motorcycle` class. Here's a possible implementation of the `Car` class:

```
class Car(object):
    """ Implementation for O.U.R. Car Dealership.
    Attributes:
        wheels: Int representing the number of wheels for the car.
        km: The integral number of kilometers driven on the car.
        make: The make of the car as a string.
        model: The model of the car as a string.
        year: The integral year the car was built.
        sold_on: The date the vehicle was sold.
    """
    def __init__(self, wheels, km, make, model, year, sold_on):
        """Return a new Car object."""
        self.wheels = wheels
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for given car as a float
amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 100000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would purchase the car."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 140000 - (.10 * self.km)
```

Looks easy, right? We could have a number of other methods in the class, but for this exercise lets focus on two methods based on our business problem: `sale_price` and `purchase_price`. Similarly, let’s have a `Truck` class:

```
class Truck(object):
    """ Implementation for O.U.R. Car Dealership.
    Attributes:
        wheels: An integer representing the number of wheels the
truck has.
```

```
        km: The integral number of kilometers driven on the truck.
        make: The make of the truck as a string.
        model: The model of the truck as a string.
        year: The integral year the truck was built.
        sold_on: The date the vehicle was sold.
    """
    def __init__(self, wheels, km, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for given truck as a float
amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 100000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would purchase the
truck."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 100000 - (.10 * self.km)
```

Look at the codes for above two classes – they look exactly the same. The functions sale_price() is exact same. Purchase_price() has different calculation logic. We can surely avoid ourselves from repeating the codes. This takes us to the next topic of Abstract classes. Lets see how to use Abstract classes for inheritance.

ABSTRACT CLASSES

A `Vehicle` is not a real-world object. Rather, it is a concept that some real-world objects (like cars, trucks, and motorcycles) embody. We would like to use the fact that each of these objects can be considered a vehicle to remove repeated code. We can do that by creating a `Vehicle` class:

```
class Vehicle(object):
    """ Implementation for O.U.R. Car Dealership.
    Attributes:
        wheels: Int for the number of wheels the vehicle has.
        km: The integral number of kilometer driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """
    base_sale_price = 0

    def __init__(self, wheels, km, make, model, year, sold_on):
        """Return a new Vehicle object."""
        self.wheels = wheels
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
    def sale_price(self):
        """Return the sale price for given vehicle as a float"""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 100000.0 * self.wheels
    def purchase_price(self):
        """Return the price to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.km)
```

Now we can make the `Car` and `Truck` class inherit from the `Vehicle` class by replacing `object` in the line `class Car(object)`. The class in parenthesis is the class that is inherited from (`object` essentially means "no inheritance". We'll discuss exactly why we write that in a bit).
We can now define `Car` and `Truck` in a very straightforward way:

```
class Car(Vehicle):
    def __init__(self, wheels, km, make, model, year, sold_on):
        """Return a new Car object."""
        self.wheels = wheels
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 140000
class Truck(Vehicle):
    def __init__(self, wheels, km, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 180000
```

This works, but has a few problems. First, we're still repeating a lot of code. We'd ultimately like to get rid of all repetition. Second, and more problematically, we've introduced the `Vehicle` class, but should we really allow people to create `Vehicle` objects (as opposed to `Car`s or `Truck`s)? A `Vehicle` is just a concept, not a real thing, so what does it mean to say the following:

```
v = Vehicle(4, 0, 'Honda', 'Accord', 2014, None)
print(v.purchase_price())
```

A `Vehicle` doesn't have a `base_sale_price`, only the individual child classes like Car and Truck do. The issue is that Vehicle should really be an Abstract Base Class. Unlike the previous example, where we were able to create object for School class, in this example, we want to restrict people from creating objects for Vehicle class. That's the reason we need to declare Vehicle class as Abstract. Abstract Base

Classes (ABC) are classes that are only meant to be inherited from; you can't create instance of an ABC. That means that, if `Vehicle` is an ABC, the following is illegal:

```
v = Vehicle(4, 0, 'Honda', 'Accord', 2014, None)
```

Do we make a class an ABC? Simple! The abc module contains a metaclass called ABCMeta). Setting a class's metaclass to ABCMeta and making one of its methods virtual makes it an ABC. A virtual method is one that the ABC says must exist in child classes, but doesn't necessarily actually implement. Now, since `vehicle_type` is an `abstractmethod`, we can't directly create an instance of `Vehicle`. As long as `Car` and `Truck` inherit from `Vehicle` and define `vehicle_type`, we can instantiate those classes just fine. The Vehicle class may be defined as follows:

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    """ Implementation for O.U.R. Car Dealership.
    Attributes:
        wheels: int the number of wheels the vehicle has.
        km: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """
    __metaclass__ = ABCMeta
    base_sale_price = 0
    wheels = 0

    def __init__(self, km, make, model, year, sold_on):
        self.km = km
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this vehicle as a float."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 100000.0 * self.wheels
    def purchase_price(self):
        """Return the price to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.km)
    @abstractmethod
    def vehicle_type():
        """"Return a string for the type of vehicle this is."""
        pass
```

Now the `Car` and `Truck` classes become:

```
class Car(Vehicle):
    """ Implementation for O.U.R. Car Dealership."""
    base_sale_price = 140000
    wheels = 4

    def vehicle_type(self):
        """"Return a string for the type of vehicle this is."""
        return 'car'

class Truck(Vehicle):
    """ Implementation for O.U.R. Car Dealership."""
    base_sale_price = 180000
    wheels = 4

    def vehicle_type(self):
        """"Return a string for the type of vehicle this is."""
        return 'truck'
```

The only difference between a car and truck is the base sale price. Defining a `Motorcycle` class, then, is similarly simple:

```
class Motorcycle(Vehicle):
    """ Implementation for O.U.R. Car Dealership."""
    base_sale_price = 80000
    wheels = 2
    def vehicle_type(self):
        """"Return a string for the type of vehicle this is."""
        return 'motorcycle'
```

Lets run the program and see the output:

```
### Lets call them
#'km', 'make', 'model', 'year', and 'sold_on'
c1 = Car(km=0,make="Tata", model = "Tigor", year = 2020,
sold_on=None)
print("Car 1 Vehicle type: ",c1.vehicle_type())
if c1.sale_price()==0:
    print("Car is already sold!")
else:
    print("Selling Price of the car is: ",c1.sale_price())
if c1.purchase_price()==0:
    print("Car is not yet sold!")
else:
    print("Purchase Price of the car is: ",c1.purchase_price())

#Creating another object of Car
c2 = Car(km=100, make="Tata", model="Tigor", year=2020,
sold_on="10-Jan-2016")
print("Car 2 Vehicle type: ", c2.vehicle_type())
if c2.sale_price() == 0:
    print("Car is already sold!")
else:
    print("Selling Price of the car is: ", c2.sale_price())
if c2.purchase_price() == 0:
    print("Car is not yet sold!")
else:
    print("Purchase Price of the car is: ", c2.purchase_price())
#Similarly, objects for Truck and Motorcycle can be created
```

```
#t = Truck()
#m = Motorcycle()
```

Hopefully, I have been able to convey a lot about what Python classes are, why they're useful, and how to use them. The topic of classes and Object-oriented Programming are deep and they reach to the core of modern programming. This chapter introduces you to the concept and shows you how to write your program in Python. You are encouraged to read more about the concepts.

Class Programs (23 July 2020)

Input Marks of 5 subjects and calculate Sum, Avg and Grade

```
import os
class Marks:
    def __init__(self,name,dept):
        self.name = name
        self.dept = dept
        self.m=-99
        self.sum=-99
        self.avg=-99
    def input_marks(self):
        self.sum=0
        for i in range(5):
            self.m = int(input("Enter the marks in subject {}:
".format(i+1)))
            self.sum+=self.m

    def display_grade(self):
        if self.sum==-99:
            print("Please call the Input function before running this")
        else:
            self.avg = self.sum/5
            #Clear the screen (in Windows)
            os.system("cls")
            print()
            #Creating Center aligned header
            print("{:^50}".format("Report Card"))
            print("Name: {} \t\t\t\t\t Department: {}".format(self.name,
self.dept))
            print("Your total marks is {} and the average is
{:.1f}".format(self.sum, self.avg))
            if self.avg >70:
                print("You got grade A")
            elif self.avg > 50:
                print("You got grade B")
            else:
                print("You got grade C")

s1= Marks("Sachin","CSE")
s1.input_marks()
s1.display_grade()
```