MODULE 4

- **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
- Let's take an example:

  Parrot is an object,

  - name, age, colour are attributes
  - singing, dancing are behaviour
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

OOP TERMINOLOGY

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behaviour to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

PILLARS OF OOP

1. **ABSTRACTION**
   - Abstraction is the process of showing only essential/necessary features of an entity/object to the outside world and hide the other irrelevant information.
   - In programming language we achieve the abstraction through public and private access modifiers and a class. So in a class make things (feature) which we want to show as public

and thing which are irrelevant make them as private so they won't be available to the outside world.

- **Real Life Example of Abstraction:**

  Real life example of Abstraction could be the gears of bike. Do we know what happens inside the engine when we change the gear? Answer is No (what happens inside the engine when we change the gear is irrelevant information from user perspective so we can hide that information). What important from users perspective is whether gear has been changed or not. (This is essential/necessary feature that must be shown to the user).

  So the abstraction says only expose that details which really matters from users perspective and hide the other details.

- **Benefits/Advantages of Using Abstraction:**

  Advantage of Abstraction is, it reduces the complexity of end users since to the end user we have shown only things that are necessary to him and not shown the unnecessary things. Imagine what would have happened when we told the bike riders things happening inside the engine (i.e. irrelevant information) when he change the gear. So it would have increased the complexity of bike riders.

- **Implementation in Programming Language**

  Imagine a car. Now let's think in terms of car rider or a person who is riding a car. So to drive a car what a car rider should know and what not.

  Necessary things:

  Brakes
  Gear
  Steering

  Unnecessary things:

  Exhaust System
  What happen when he change the Gear of car.
  Silencer

2. ENCAPSULATION

- Encapsulation is a process of binding data members (variables, properties) and member functions (methods) together. In object oriented programming language we achieve encapsulation through Class.
- **Real Life Example of Encapsulation:**

  The real life example of encapsulation will be the Capsule. Capsule binds all chemical contents required for curing specific disease together just like the class which binds data members and member functions.

- **Benefits/Advantages of Using Encapsulation:**
  - **Encapsulation as Information Hiding Mechanism:**

    As already discussed, encapsulation is a process of binding data members (variables, properties) and member functions (methods) together. Due to this approach the contents (or data members) of object is generally hidden from the

outside world (By simply making the data members as private) which is nothing but information hiding.

Typically, only the objects own methods can directly inspect or manipulate its fields. So the outside world only knows what a class does (Through the member functions) but don't know what are its data members and how that class do a specific task.

Hiding the internals of the object or allowing access to it through only member function protects objects integrity by preventing users from setting the internal data or the component in such a way that may lead to an invalid or inconsistent state of object.

3. INHERITANCE

- The process of creating the new class by extending the existing class is called inheritance or the process of inheriting the features of base class is called as inheritance.
- The existing class is called the base class and new class which is created from it is called the derived class.
- **Real Life Example of Inheritance:**

  We inherit some of our features (may be body color, nose shape, height of body etc) from Mom and Dad.
- **Benefits/Advantages of Using Inheritance:**

  The most important advantage of inheritance is code re-usability. In Inheritance the derived class possess all attributes/properties and functions of base class, this is where code re-usability comes into the picture. So no need to write same function and attributes of base class in a derived class.

  Note: Private members of base class also get inherited in derived class but they are not accessible in derived class.

4. POLYMORPHISM

- Poly means many and Morph means forms. Polymorphism is the process in which an object or function take different forms.
- **Real Life Example Of Polymorphism:**

  Real life example of Polymorphism is mobile phone. It is a single object but it can be used for making calls, listening music, sending mails, taking pictures, etc (different forms).

DIFFERENCE BETWEEN PROCEDURE ORIENTED PROGRAMMING (POP) AND OBJECT ORIENTED PROGRAMMING (OOP)

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, Python, VB.NET, C#.NET.

CLASSES, OBJECTS, METHODS AND FUNCTIONS

1.1 CLASSES AND OBJECTS

- Classes and objects are two main aspects of object oriented programming language.
- **A class creates a new type and object is an instance (or variable) of the class.**
- **Classes provides a blueprint or a template using which objects are created.**
- In Python, everything is an object or an instance of some class. For example, all integer variables that we define in our program are actually instances of class int. Similarly, all string variables are objects of class string.
- The Python Standard Library is based on the concept of classes and objects.
- The general syntax for defining a class is:

```
class class_name:
      <statement-1>
      <statement-2>
      .
      .
      .
      <statement-N>
```

- The class definition is similar to function definition. It starts with a keyboard class followed by the class_name and a colon (:). The statement in the definition can be any of these – sequential instructions, decision control statements, loop statements, and even include function definitions.
- **Variables defined in a class are called class variables and functions defined inside a class are called class methods. Class variables and class methods together are known as class members.**
- The class members can be accessed through class objects. Class methods have access to all the data contained in the instance of the object.
- Class definitions can appear anywhere in a program, but they are usually written near the beginning of the program, after the import statements.

1.1.1 CREATING OBJECTS

- Once a class is defined, the next job is to create an object (or instance) of that class. **The object can then access class variables and class methods using the dot operator (.).**
- The syntax to create an object is:

```
object_name = class_name()
```

- Creating an object or instance of a class is known as class instantiation. From the syntax, we can see that class instantiation uses function notation. Using the syntax, an empty object of a class is created. Thus, we see that in Python, to create a new object, call a class as if it were a function.
- The syntax for accessing a class member through the class object is:

```
object_name.class_member_name
```

- **For example:**

```
class Demo:
  var = 10
```

```
obj = Demo()
print(obj.var)
```

Output:

10

1.1.2 DATA ABSTRACTION AND HIDING THROUGH CLASSES

- **Data abstraction refers to the process by which data and functions are defined in such a way that only essential details are provided to the outside world and the implementation details are hidden.**
- In Python and other object oriented programing languages, classes provide methods to the outside world to provide functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.
- **Data encapsulation, also called as data hiding, organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.**
- Encapsulation defines different access levels for data variables and member function of the class. These access levels specifies the access rights, for example:
  - Any data or function with access level public can be accessed by any function belonging to any class. This is the lowest level of data protection.
  - Any data or function with access level private can be accessed only by the class in which it is declared. This is the highest level of data protection. In Python, private variables are prefixed with a double underscore (__). For example, __var is a private variable of the class.

1.2 CLASS METHOD AND SELF ARGUMENT

- Class methods (or functions defined in the class) are exactly same as ordinary functions that we have been defining+so far with just one small difference.
- **Class methods must have the first argument named as self.**
- This is the first argument that is added to the beginning of the parameter list. We don't have to pass a value for this parameter when you call the method. Python provides its value automatically.
- The self argument refers to the object itself. That is, the object that has called the method. This means that even if a method that takes no arguments, it should be defined to accept the self. Similarly, a function defined to accept one parameter will actually take two – self and the parameter, so on and so forth.
- Since, the class methods uses self, they require an object or instance of the class to be used. For this reason, they are often referred to as **instance methods**.
- **Note:**
  - **The statements inside the class definition must be properly intended.**
  - **A class that has no other statements should have a pass statement at least.**

- **Class methods or functions that begins with double underscore (__) are special functions with a predefined and a special meaning.**

➢ **For example:**

```
class selfarg:
   var = 10
   def display(self):
      print('In class method....')

obj = selfarg()
print(obj.var)
obj.display()
```

Output:

```
10
In class method....
```

1.3 THE __init__() METHOD (THE CLASS CONSTRUCTOR)

➢ **The __init__() method is automatically executed when an object of a class is created. The method is useful to initialize the variables of the class object.**

➢ **For example:**

```
class Demo:
   def __init__(self, val):
      print('In class method....')
      self.val = val
      print('The value is: ', val)

obj = Demo(10)
```

Output:

```
In class method....
The value is:  10
```

➢ The __init__() method accepts one argument val. Like any other class method the first argument has to be self. In the __init__() method we define a variable as self.val which has exactly the same name as that specified in the argument list. Though the two variables have the same name, they are entirely different variables. The self.val belongs to the newly created object. Note that we have just created an object in the main module and no where we have called the __init__() method. This is because the __init__() method is automatically called when the object of the class is created.

1.4 CLASS VARIABLES AND OBJECT VARIABLES

➢ Class variables are owned by the class and object variables owned by each object.

- If a class has n objects, then there will be n separate copies of the object variable as each object will have its own object variable.
- The object variable is not shared between objects.
- A change made to the object variable by one object will not be reflected in other objects.
- If a class has one class variable, then there will be one copy for that variable. All the objects of that class will share the class variable.
- Since there exists a single copy of the class variable, any change made to the class variable by an object will be reflected in all other objects.

For example:

```
class obclvar:
   class_var = 0
   def __init__(self,var):
      obclvar.class_var += 1
      print('The object value is: ',var)
      print('The class variable is: ',obclvar.class_var)

obj = obclvar(10)
obj1 = obclvar(20)
obj2 = obclvar(30)
```

Output:

```
The object value is:  10
The class variable is:  1
The object value is:  20
The class variable is:  2
The object value is:  30
The class variable is:  3
```

1.5 THE __del__() METHOD

- __del__() method does the opposite of __init__() method.
- **The __del__() method is automatically called when an object is going out of scope. This is the time when an object will no longer be used and its occupied resources are returned back to the system so that they can be reused as and when required.**
- **For example:**

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

   def __del__(self):
      print('The object is deleted')
```

```
obj = Demo(10)
print(obj)
del obj
print(obj)
```

Output:

```
<__main__.Demo object at 0x058B3BB0>
The object is deleted
Traceback (most recent call last):
  File "C:\Users\akhil\AppData\Local\Programs\Python\Python36-32\classex.py", line 371, in <module>
    print(obj)
NameError: name 'obj' is not defined
```

1.6 THE __str__() METHOD

- **__str__() method returns a string representation of an object.**
- When we print an object, Python invokes the str method:

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

  def __str__(self):
    return '%d' %self.x


obj = Demo(10)
print(obj)
```

Output:

```
10
```

1.7 OPERATOR OVERLOADING

- The meaning of operators like +, =, *, /, >, <, etc. are pre-defined in any programming language. Programmers use them directly on built-in data types to write their own programs. But, for user-defined data types like objects, these operators do not work.
- **Python allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called operator overloading which allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user-defined data types.**
- With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator. This enables the programmer to perform some

specific computation when the operator is applied on class objects and to apply a standard definition when the same operator is applied on a built-in data type.

- While evaluating an expression with operators, Python looks at the operands around the operator. If the operands are of built-in types, Python calls a built-in routine. In case, the operator is being applied on user-defined operand(s), the Python compiler checks to see if the programmer has an overloaded operator function that it can call. If such a function whose parameters match the type(s) and number of the operands exists in the program, the function is called, otherwise a compiler error is generated.
- Operator overloading is a form of compile time polymorphism.
- **Advantages of operator overloading:**
  - Programmers can use the same notations for user-defined objects and built-in objects. Example, to add two complex numbers, we can simply write C1 + C2.
  - With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to built-in types.
- **Table: Operators and their corresponding function names**

Operator	Function Name
+	__add__
-	__sub__
*	__mul__
/	__truediv__
**	__pow__
%	__mod__
>>	__rshift__
&	__and__
|	__or__
^	__xor__
~	__invert__
<<	__lshift__
>	__gt__
<	__lt__
>=	__ge__
+=	__iadd__
-=	__isub__
*=	__imul__
/=	__idiv__
**=	__ipow__
%=	__imod__
>>=	__irshift__
&=	__iand__
|=	__ior__
^=	__ixor__
~=	__iinvert__
<<=	__ilshift__
<=	__le__

==	__eq__
!=	__ne__

- **Program to add two complex numbers without overloading the + operator**

```
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0

    def setvalues(self, real, imag):
        self.real = real
        self.imag = imag

    def display(self):
        print("The result is:",self.real,"+",self.imag,"i")

C1 = Complex()
C1.setvalues(1,2)

C2 = Complex()
C2.setvalues(3,4)

C3 = Complex()

C3 = C1+C2

C3.display()
```

Output:

```
Traceback (most recent call last):
  File  "C:\Users\akhil\AppData\Local\Programs\Python\Python36-32\classex.py",  line 474, in <module>
    C3 = C1+C2
TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

- \+ Operator does not work on user-defined objects. Now, to do the same concept, we will add an operator overloading function in the code.
- **Program to add two complex numbers with overloading the + operator**

```
class Complex:
    def __init__(self):
        self.real = 0
        self.imag = 0
```

```
    def setvalues(self, real, imag):
        self.real = real
        self.imag = imag

    def display(self):
        print("The result is:",self.real,"+",self.imag,"i")

    def __add__(self,C):
        T = Complex()
        print(self.real,self.imag)
        print(C.real,C.imag)
        T.real = self.real+C.real
        T.imag = self.imag+C.imag
        return T

C1 = Complex()
C1.setvalues(1,2)

C2 = Complex()
C2.setvalues(3,4)

C3 = Complex()

C3 = C1+C2

C3.display()
```

Output:

```
1 2
3 4
The result is: 4 + 6 i
```

- When we write C1 + C2, the __add__() function is called on C1 on C2 is passed as an argument. Remember that, user-defined classes have no + operator defined by default.

1.8 INHERITANCE

- Reusability is an important feature of OOP. I saves efforts and cost required to build a software product, but also enhances its reliability.
- It also doesn't require to re-write, re-debug and re-test the code.
- To support reusability, Python supports the concept of re-using existing classes.

- Python allows its programmers to create new classes that re-use the re-written and tested classes. The existing classes are adapted as per user's requirements so that the newly formed classes can be incorporated in current software application being developed.
- **The technique of creating a new class from an existing class is called inheritance. The old or existing class is called the base class and the new class in known as the derived class or subclass.**
- The derived classes are created by first inheriting the data and methods of the base class and then adding new specialised data and functions in it. In this process of inheritance, the base class remains unchanged.
- The derived class inherits all the capabilities of the base class and adds refinements and extensions of its own.
- The concept of inheritance is therefore, frequently used to implement the 'is-a' relationship.
- For example: teacher is-a person, student is-a person; while both teacher and student are a person in the first place, both also have some distinguishing features. So all the common traits of a teacher and student are specified in the Person class and specialized features are incorporated in two separate classes – Teacher and Student.
- The concept of inheritance follows a top-down approach o problem solving. In top-down approach, generalized classes are designed first and then specialized classes are derived by inheriting/extending the generalized classes.
- The syntax to inherit a class:

```
Class DerivedClass(BaseClass):
    Body_of_derived_class
```

- **For example:**

```
#Super Class
class Person:
  def __init__(self,first,last,age):
    self.first=first
    self.last=last
    self.email=first+"_"+last+"@gmail.com"
    self.age=age
  def display(self):
    print("NAME : ",self.first,self.last)
    print("AGE : ",self.age)
    print("E-MAIL : ",self.email)

#Sub-Class1
class Teacher(Person):
  def __init__(self,first,last,age,exp,r_area):
    Person.__init__(self,first,last,age)
    #super().__init__(first,last,age)
    self.exp=exp
    self.r_area=r_area
  def displayData(self):
```

```
        Person.display(self)
        print("YEARS OF EXPERIENCE : ",self.exp)
        print("RESEARCH AREA : ",self.r_area)

#Sub-Class2
class Student(Person):
    def __init__(self,first,last,age,course,marks):
        Person.__init__(self,first,last,age)
        self.course=course
        self.marks=marks
    def displayData(self):
        Person.display(self)
        print("COURSE : ",self.course)
        print("MARKS : ",self.marks)

print("**********TEACHER**********")
T=Teacher("Akhilaa","Reddy",27,5,"Networking")
T.displayData()

print("**********STUDENT**********")
S=Student("Nikkhil","D",21,"B.E.",99)
S.displayData()
```

Output:

```
**********TEACHER**********
NAME :  Akhilaa Reddy
AGE :  27
E-MAIL :  Akhilaa_Reddy@gmail.com
YEARS OF EXPERIENCE :  5
RESEARCH AREA :  Networking
**********STUDENT**********
NAME :  Nikkhil D
AGE :  21
E-MAIL :  Nikkhil_D@gmail.com
COURSE :  B.E.
MARKS :  99
```

1.9 POLYMORHISM AND METHOD OVERRIDING

- **Polymorphism refers to having several different forms. It enables the programmers to assign a different meaning or usage to a variable, function, or an object in different contexts.**

- While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, it is related to methods.
- When polymorphism is applied to a function or method depending on the given parameters, a particular form of the function can be selected for execution. In Python, method overriding is one way of implementing polymorphism.
- In the above program, the __ini__() method was defined in all the three classes. When this happens, the method in the derived class overrides that in the base class. This means that __init__() in Teacher and Student gets preference over the __init__() method in the Person class. Thus, method overriding is the ability of a class to change the implementation of a method provided by one of its ancestors.
- Another thing is that when we override a base class method, we extend the functionality of the base class method. This is done by calling the method in the base class method from the derived class method and also adding additional statements in the derived class method.

1.10 USER-DEFINED TYPES

- Consider an example where we want create a type called Point that represents a point in two-dimensional space.
- In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (*x*, *y*) represents the point *x* units to the right and *y* units up from the origin.
- There are several ways we might represent points in Python:
  - We could store the coordinates separately in two variables, x and y.
  - We could store the coordinates as elements in a list or tuple.
  - We could create a new type to represent points as objects
- **A user-defined type is called a class.**
- A class can be defined as follows:

  For example:

```
class Point:
    """Represents a point in 2-D space."""
    pass
```

- **A class has two parts:**
  - **Header – indicates a new class which is kind of an object and which is built-in type.**
  - **Body – is a docstring that explains what the class is for. We can define variables and functions inside the class.**
- Defining a class named Point creates a class object.

```
>>> print(Point)
<class '__main__.Point'>
```

- Because Point is defined at the top level, its "full name" is __main__.Point.
- The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function.

  For example:

```
>>> blank = Point()
>>> print blank
<__main__.Point object at 0x03453BB0>
```

- The return value is a reference to a Point object, which we assign to blank. Creating a new object is called **instantiation**, and the object is an **instance** of the class.
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

1.11 ATTRIBUTES

- We can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

- This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.
- The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram.**

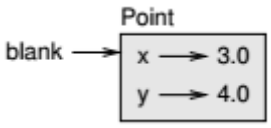

Fig: Object diagram

- The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.
- We can read the value of attribute as follows:

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

- The expression blank.x means, “Go to the object blank refers to and get the value of x.” In this case, we assign that value to a variable named x. There is no conflict between the variable x and the attribute.

1.12 RECTANGLES

- Imagine you are designing a class to represent rectangles.
- There are at least two possibilities:
  - You could specify one corner of the rectangle (or the center), the width, and the height.
  - You could specify two opposing corners.
- We will be implementing the first one.

- For example:

```
class Rectangle:
    """ Represents a rectangle.
    attribue: width, height, corner.
    """
    pass
```

- The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.
- To represent a rectangle, we have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
>>> box.width = 100.0
>>> box.height = 200.0
>>> box.corner = Point()
>>> box.corner.x = 0.0
>>> box.corner.y = 0.0
```

- The expression box.corner.x means, "Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x."
- The below figure shows the state of this object. An object that is an attribute of another object is **embedded**.

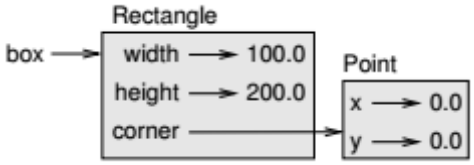

Fig: Object Diagram

1.13 INSTANCES AS RETURN VALUES

- Functions can return instances. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
class Point:
   pass

class Rectangle:
   def find_center(self,rect):
      p = Point()
      p.x = rect.corner.x + rect.width/2.0
      p.y = rect.corner.y + rect.height/2.0
      return p

box = Rectangle()
```

```
box.width = 100.0
box.height = 200.0

box.corner = Point()
box.corner.x = 0.3
box.corner.y = 1.3

center = box.find_center(box)
print(center.x, center.y)
```

Output:

```
50.3 101.3
```

1.14 COPYING

- Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.
- Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
import copy

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

p1 = Point(3.0,4.0)

p2 = copy.copy(p1)

print(p1 is p2)

print(p1 == p2)

print(id(p1), id(p2))
```

Output:

```
False
False
84491184 97151664
```

- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. The

default behaviour of the == operator is the same as the is operator; it checks object identity, not object equivalence.

- If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
import copy

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

class Rectangle:
   def __init__(self,width,height):
      self.width = width
      self.height = height

box1 = Rectangle(100.0,200.0)

box1.corner = Point(0.0,0.0)

box2 = copy.copy(box1)

print(box1 is box2)

print(box1.corner == box2.corner)

print(id(box1), id(box2))
```

Output:

```
False
True
88554416 101405104
```

- This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

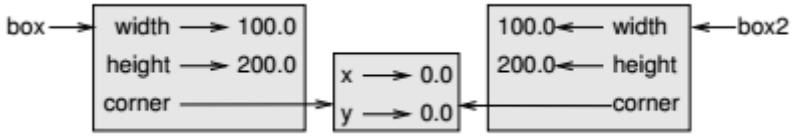

Fig: Object Diagram

- For most applications, this is not what you want. the copy module contains a method named deepcopy that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. This is called a **deep copy**.

```
import copy

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

class Rectangle:
   def __init__(self,width,height):
      self.width = width
      self.height = height

box1 = Rectangle(100.0,200.0)

box1.corner = Point(0.0,0.0)

box3 = copy.deepcopy(box1)

print(box3 is box1)

print(box3.corner == box1.corner)
```

Output:

False
False

shallow copy	To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.
deep copy	To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

1.15 EXERCISES

1. Write a function called distance_between_points that takes two Points as arguments and returns the distance between them.

```
import math

class Point:
   def distance_between_points(self,p1, p2):
      d = math.sqrt(math.pow((p2.x - p1.x),2)+math.pow((p2.y - p1.y),2))
      return d
```

```
p1 = Point()
p1.x = 3
p1.y = 2

p2 = Point()
p2.x = 9
p2.y = 7

res = p1.distance_between_points(p1,p2);
print(res)
```

Output:

7.810249675906654

2. Write a program that has class Time with attributes hour, minute, seconds. Write a program to add two objects.
3. Write a function named move_rectangle that takes a Rectangle and two numbers named dx and dy. It should change the location of the rectangle by adding dx to the x coordinate of corner and adding dy to the y coordinate of corner.
4. Write a function called print_time that takes a Time object and prints it in the form hour:minute:second. Hint: the format sequence '%.2d' prints an integer using at least two digits, including a leading zero if necessary.

```
class Time:
  """Represents the time of day.
  attributes: hour, minute, second"""

time = Time()
time.hour = 11
time.minute = 59
time.second = 30

def print_time(time):
  print "%.2d:%.2d:%.2d" % (time.hour, time.minute, time.second)

print_time(time)
```

5. Write a boolean function called is_after that takes two Time objects, t1 and t2, and returns True if t1 follows t2 chronologically and False otherwise.

```
class Time(object):
  """Represents the time of day.
  attributes: hour, minute, second"""

time = Time()
```

```
time.hour = 11
time.minute = 59
time.second = 30

time2 = Time()
time2.hour = 11
time2.minute = 59
time2.second = 35

def is_after(t1, t2):
  return (t1.hour, t1.minute, t1.second) > (t2.hour, t2.minute, t2.second)
```

6. Use datetime module:
   I. Use the datetime module to write a program that gets the current date and prints the day of the week.

```
import datetime

def print_date_and_day_of_week():
  date = datetime.date.today()
  days_of_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
  day_of_week = days_of_week[date.weekday()]
  print(date, " : ", day_of_week)

print_date_and_day_of_week()
```

Output:

2018-05-03 : Thursday

   II. Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.

```
import datetime

def time_to_next_birthday(birth_year, birth_month, birth_day):
  birth = datetime.datetime(birth_year, birth_month, birth_day)
  current_time = datetime.datetime.now()

  current_age = current_time.year - birth.year - ((current_time.month,current_time.day) < (birth.month, birth.day))

  if (current_time.month, current_time.day) < (birth.month, birth.day):
    next_bday = (current_time.year, birth_month, birth_day)
  else:
```

```
        next_bday = (current_time.year + 1, birth_month, birth_day)

    time_to_next_bday = datetime.datetime(*next_bday) - current_time

    print ("current age: ", current_age)
    print ("time to next birthday: ", time_to_next_bday)
```

III. For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birthdays and computes their Double Day.

```
import datetime

def find_double_day(bday1, bday2):
    """bday1 is the older person.
    both birthdays are entered as tuples in this format:
    (year, month, day)"""

    person1 = datetime.date(*bday1)
    person2 = datetime.date(*bday2)

    age_diff = -(person1 - person2)

    p1 = int(age_diff.days)
    p2 = 0

    while p2 * 2 != p1:
        p1 += 1
        p2 += 1

    date_at_twice_age = person2 + datetime.timedelta(days=p2)
    print (date_at_twice_age, "\n", "person 1 was %d days old, and person 2 was %d days
    old" % (p1, p2))
```

IV. For a little more challenge, write the more general version that computes the day when one person is n times older than the other.

```
import datetime

def find_n_times_day(bday1, bday2, n):
    """bday1 is the older person.
    both birthdays are entered as tuples in this format:
    (year, month, day)"""

    person1 = datetime.date(*bday1)
```

```
person2 = datetime.date(*bday2)

age_diff = -(person1 - person2)

p1 = int(age_diff.days)
p2 = 0

while p2 * n != p1:
   if p2 * n > p1:
      print "This never precisely occurred"
      return None
   p1 += 1
   p2 += 1

date_at_n_times_age = person2 + datetime.timedelta(days=p2)

print (date_at_n_times_age, "\n", "person 1 was %d days old, and person 2 was %d
days old" % (p1, p2))
```

7. Write a program that has a class Circle. Use a class variable to define the value of constant PI. Use this variable to calculate area and circumference of a circle with specified radius.

```
class Circle:
   PI = 3.1412
   def __init__(self,radius):
      self.radius = radius

   def area(self):
      ar = Circle.PI * self.radius *self.radius
      print('Area of circle is',ar)

   def circumference(self):
      cr = 2 * Circle.PI * self.radius
      print('Circumference of a Circle is', cr)

c = Circle(2.5)
c.area()
c.circumference()
```

Output:

```
Area of circle is 19.6325
Circumference of a Circle is 15.706
```

8. Write a program with class Employee that keeps a track of the number of employee in an organization and also stores their name, designation and salary.

```
class employee:
    count=0
    def input(self):
        self.name=input('Enter name:')
        self.des=input('Enter designation:')
        self.salary=float(input('Enter salary:'))
        employee.count=employee.count+1
    def display(self):
        print('count of emp. :',employee.count)
    def details(self):
        print('name:' ,self.name,'\n designation:',self.des,'\n salary:',self.salary)

a1 = employee()
a1.input()

a2 = employee()
a2.input()

a1.display()
a1.details()
```

Output:

```
Enter name:akhilaa
Enter designation:assistant professor
Enter salary:50000
Enter name:arnav singh
Enter designation:professor
Enter salary:150000
count of emp. : 2
name: akhilaa
 designation: assistant professor
 salary: 50000.0
```

9. Write a program that uses class to store the name and marks of students. Use list to store the marks in three subjects.

```
class student:
    def __init__(self,name):
        self.name=name
        self.marks=[]
    def Enter_marks(self):
        for i in range(0,3):
            print('Enter the marks:')
            m=int(input())
```

```
            self.marks.append(m)
        def Display(self):
            print('Name is: ',self.name,'\n','Marks :',self.marks)
    name=input('Enter the name: ')
    s=student(name)
    s.Enter_marks()
    s.Display()
```

Output:

```
Enter the name: akhilaa
Enter the marks:
90
Enter the marks:
99
Enter the marks:
87
Name is:  akhilaa
 Marks : [90, 99, 87]
```

10. Write a program that has a class Person storing name and date of birth (DOB) of a person. The program should subtract the DOB from today's date to find out whether a person is eligible to vote or not.

```
import datetime

class Person:
    def __init__(self,name,dob):
        self.name = name
        self.dob = dob

    def check(self):
        today = datetime.date.today()
        age = today.year - self.dob.year
        if(age>=18):
            print('You are eligible to vote!!!')
        else:
            print('You are not eligible to vote!!!')

P = Person('akhilaa',datetime.date(1990,12,1))
P.check()
```

Output:

```
You are eligible to vote!!!
```

11. Write a program to find the day of the week.

```
import datetime

date = datetime.date.today()
days_of_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
day_of_week = days_of_week[date.weekday()]
print(date, " : ", day_of_week)
```

Output:

```
2018-05-15 : Tuesday
```

12. Write a program to deposit or withdraw money in a bank account.

1.16 GLOSSARY

class	A user-defined type. A class definition creates a new class object.
class object	An object that contains information about a user-defined type. The class object can be used to create instances of the type.
instance	An object that belongs to a class.
attribute	One of the named values associated with an object.
embedded (object)	An object that is stored as an attribute of another object.
shallow copy	To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.
deep copy	To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.
object diagram	A diagram that shows objects, their attributes, and the values of the attributes.

QUESTIONS

1. What are classes and objects? Explain with syntax and example how to create them.
2. Explain the 4 pillars of object oriented programming.
3. What is self-argument?
4. Difference between class variables and object variables.
5. Explain with an example:
   a. __init__()
   b. __str__()
   c. __del__()
6. Explain shallow copy and deep copy with example.
7. What is operator overloading? Explain how '+' operator can be overloading with an example.
8. What is inheritance? Explain with an example.