## 4.1 CLASSES AND OBJECTS

Python is an object-oriented programming language, and *class* is a basis for any object oriented programming language. Class is a user-defined data type which binds data and functions together into single entity. Class is just a prototype (or a logical entity/blue print) which will not consume any memory. An object is an instance of a class and it has physical existence. One can create any number of objects for a class. A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

### 4.1.1 Programmer-defined Types

A class in Python can be created using a keyword `class`. Here, we are creating an empty class without any members by just using the keyword `pass` within it.

```
class Point:
    pass

print(Point)
```

The output would be –

```
<class '__main__.Point'>
```

The term `__main__` indicates that the class `Point` is in the main scope of the current module. In other words, this class is at the top level while executing the program.

Now, a user-defined data type `Point` got created, and this can be used to create any number of objects of this class. Observe the following statements –

```
p=Point()
```

Now, a reference (for easy understanding, treat reference as a pointer) to `Point` object is created and is returned. This returned reference is assigned to the object `p`. The process of creating a new object is called as ***instantiation*** and the object is ***instance*** of a class. When we print an object, Python tells which class it belongs to and where it is stored in the memory.

```
print(p)
```

The output would be –

```
<__main__.Point object at 0x003C1BF0>
```

The output displays the address (in hexadecimal format) of the object in the memory. It is now clear that, the object occupies the physical space, whereas the class does not.

### 4.1.2 Attributes

An object can contain named elements known as ***attributes***. One can assign values to these attributes using dot operator. For example, keeping coordinate points in mind, we can assign two attributes `x` and `y` for the object `p` of a class `Point` as below –

```
p.x =10.0
p.y =20.0
```

A state diagram that shows an object and its attributes is called as ***object diagram.*** For the object `p`, the object diagram is shown in Figure 4.1.

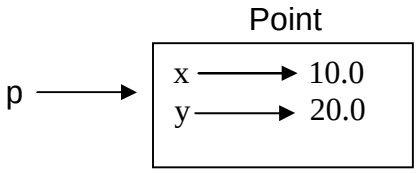



Figure 4.1 Object Diagram

***The diagram indicates that a variable (i.e. object) `p` refers to a `Point` object***, which contains two attributes. Each attributes refers to a floating point number.

One can access attributes of an object as shown –

```
>>> print(p.x)
     10.0
>>> print(p.y)
     20.0
```

Here, `p.x` means *"Go to the object p refers to and get the value of x".* Attributes of an object can be assigned to other variables –

```
>>> x= p.x
>>> print(x)
     10.0
```

Here, the variable `x` is nothing to do with attribute `x`. There will not be any name conflict between normal program variable and attributes of an object.

**A complete program:** Write a class Point representing a point on coordinate system. Implement following functions –

- A function `read_point()` to receive x and y attributes of a Point object as user input.
- A function `distance()` which takes two objects of Point class as arguments and computes the Euclidean distance between them.
- A function `print_point()` to display one point in the form of ordered-pair.

```
import math

class Point:
    """ This is a class Point representing
     a coordinate point
```

```
    """

def  read_point(p):
    p.x=float(input("x coordinate:"))
    p.y=float(input("y coordinate:"))

def print_point(p):
    print("(%g,%g)"%(p.x, p.y))

def distance(p1,p2):
    d=math.sqrt((p1.x-p2.x)**2+(p1.y-p2.y)**2)
    return d

p1=Point()                      #create first object
print("Enter First point:")
read_point(p1)                  #read x and y for p1

p2=Point()                      #create second object
print("Enter Second point:")
read_point(p2)                  #read x and y for p2

dist=distance(p1,p2)            #compute distance
print("First point is:")
print_point(p1)                 #print p1
print("Second point is:")
print_point(p2)                 #print p2

print("Distance is: %g" %(distance(p1,p2)))   #print d
```

The sample output of above program would be –

```
Enter First point:
x coordinate:10
y coordinate:20
Enter Second point:
x coordinate:3
y coordinate:5
First point is: (10,20)

Second point is:(3,5)
Distance is: 16.5529
```

Let us discuss the working of above program thoroughly –

- The class `Point` contains a string enclosed within 3 double-quotes. This is known as ***docstring***. Usually, a string literal is written within 3 consecutive double-quotes inside a class, module or function definition. It is an important part of documentation and is to help someone to understand the purpose of the said class/module/function. The docstring becomes a value for the special attribute viz.`__doc__`available for

any class (and objects of that class). To get the value of docstring associated with a class, one can use the statements like –

```
>>> print(Point.__doc__)
 This is a class Point representing a coordinate point

>>> print(p1.__doc__)
 This is a class Point representing a coordinate point
```

Note that, you need to type two underscores, then the word `doc` and again two underscores.

In the above program, there is no need of docstring and we would have just used `pass` to indicate an empty class. But, it is better to understand the professional way of writing user-defined types and hence, introduced docstring.

- The function `read_point()` take one argument of type `Point` object. W hen we use the statements like,

  ```
  read_point(p1)
  ```

  the parameter `p` of this function will act as an alias for the argument `p1`. Hence, the modification done to the alias `p` reflects the original argument `p1`. With the help of this function, we are instructing Python that the object p1 has two attributes x and y.

- The function `print_point()` also takes one argument and with the help of format-strings, we are printing the attributes x and y of the Point object as an ordered-pair (x,y).

- As we know, the Euclidean distance between two points (x1,y1) and (x2,y2) is

  $$\sqrt{(x1-x2)^2+(y1-y2)^2}$$

  In this program, we have Point objects as (p1.x, p1.y) and (p2.x, p2.y). Apply the formula on these points by passing objects p1 and p2 as parameters to the function `distance()`. And then return the result.

Thus, the above program gives an idea of defining a class, instantiating objects, creating attributes, defining functions that takes objects as arguments and finally, calling (or invoking) such functions whenever and wherever necessary.

**NOTE:** User-defined classes in Python have two types of attributes viz. ***class attributes*** and ***instance attributes***. Class attributes are defined inside the class (usually, immediately after class header). They are common to all the objects of that class. That is, they are shared by all the objects created from that class. But, instance attributes defined for individual objects. They are available only for that instance (or object). Attributes of one instance are not available for another instance of the same class. For example, consider the class Point as discussed earlier –

```
class Point:
     pass

p1= Point()                    #first object of the class
p1.x=10.0                      #attributes for p1
p1.y=20.0
print(p1.x, p1.y)              #prints 10.0 20.0

p2= Point()                    #second object of the class
print(p2.x)                    #displays error as below

AttributeError: 'Point' object has no attribute 'x'
```

This clearly indicates that the attributes x and y created are available only for the object p1, but not for p2. Thus, x and y are instance attributes but not class attributes.

We will discuss class attributes late in-detail. But, for the understanding purpose, observe the following example –

```
class Point:
     x=2
     y=3

p1=Point()                     #first object of the class
print(p1.x, p1.y)              # prints 2 3

p2=Point()                     #second object of the class
print(p2.x, p2.y)              # prints 2 3
```

Here, the attributes `x` and `y` are defined inside the definition of the class `Point` itself. Hence, they are available to all the objects of that class.

### 4.1.3 Rectangles

It is possible to make an object of one class as an attribute to other class. To illustrate this, consider an example of creating a class called as Rectangle. A rectangle can be created using any of the following data –

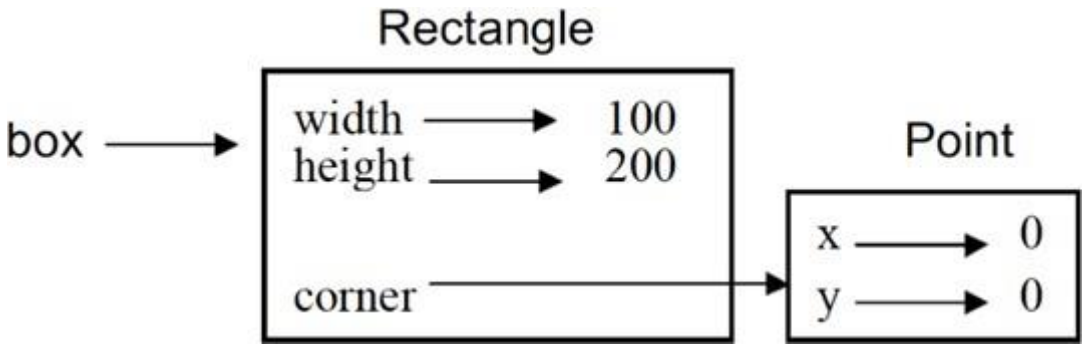
- By knowing width and height of a rectangle and one corner point (ideally, a bottom-left corner) in a coordinate system
- By knowing two opposite corner points

Let us consider the first technique and implement the task: Write a class Rectangle containing numeric attributes width and height. This class should contain another attribute *corner* which is an instance of another class Point.
The program is as given below –

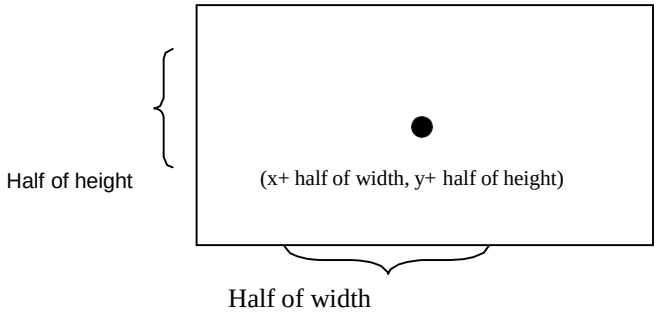
```
class Point:
    def_init_(self,x,y):
        self.x=x
        self.y=y
    def print_point(pt):
        print("({0}, {1})".format(pt.x, pt.y))

class Rectangle:

    """ A class to manufacture rectangle objects """

    def__init__(self, posn, w, h):
        """ Initialize rectangle at posn, with width w, height h """
        self.corner = posn
        self.width = w
        self.height = h


    def grow(self, delta_width, delta_height):
        """ Grow (or shrink) this object by the deltas """
        self.width += delta_width
        self.height += delta_height

    def move(self, dx, dy):
        """ Move this object by the deltas """
        self.corner.x += dx
        self.corner.y += dy

    def__str__(self):
        return "({0},{1},{2},{3})".format(self.corner.x,self.corner.y,self.width,
self.height)


def same_coordinates(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)

def midpoint(p1, p2):
        """ Return the midpoint of points p1 and p2 """
        mx = (p1.x + p2.x)/2
        my = (p1.y + p2.y)/2
        return Point(mx, my)


box1 = Rectangle(Point(0, 0), 100, 200)
box2 = Rectangle(Point(100, 80), 5, 10)
```

```
print("box1: ", box1)
print("box2: ", box2)
box1.grow(25, -10)
print(box1)
box1.move(-10, 10)
print(box1)

p1 = Point(3, 4)
p2 = Point(5, 8)
print(same_coordinates(p1, p2))

p1.print_point()

r=midpoint(p1, p2)
r.print_point()
```

A local object `p` of type `Point` has been created inside this function. The attributes of p are x and y, which takes the values as the coordinates of center point of rectangle.




Center of a rectangle can be computed with the help of following diagram.



```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

The funtion `find_center()` returns the computed center point. Note that, the return value of a function here is an instance of some class. That is, one can have an ***instance as return values*** from a function.

### 4.1.4 Copying

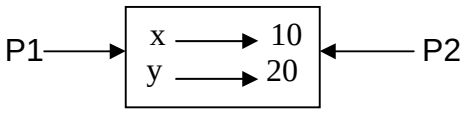
An object will be aliased whenever there an object is assigned to another object of same class. This may happen in following situations –

- Direct object assignment (like p2=p1)
- When an object is passed as an argument to a function
- When an object is returned from a function

The last two cases have been understood from the two programs in previous sections. Let us understand the concept of aliasing more in detail using the following program –

```
>>> class Point:
        pass

>>> P1=Point()
>>> P1.x=10
>>> P1.y=20
>>> P2=P1
>>> print(P1)
    <__main__.Point object at 0x01581BF0>
>>> print(P2)
    <__main__.Point object at 0x01581BF0>
```

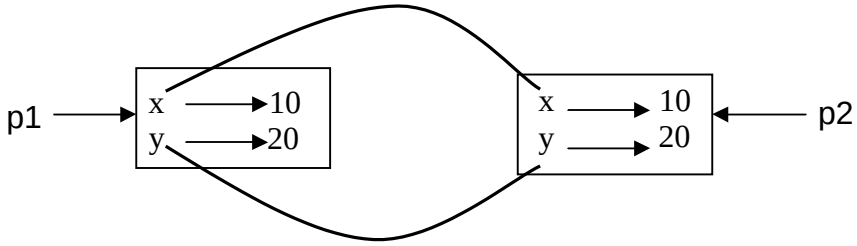
Observe that both p1 and p2 objects have same physical memory. It is clear now that the object p2 is an alias for p1. So, we can draw the object diagram as below –




Hence, if we check for equality and identity of these two objects, we will get following result.

```
>>> p1 is p2
    True
>>> p1==p2
    True
```

But, the aliasing is not good always. For example, we may need to create a new object using an existing object such that – the new object should have a different physical memory, but it must have same attribute (and their values) as that of existing object. Diagrammatically, we need something as below



In short, ***we need a copy of an object, but not an alias.*** To do this, Python provides a module called ***copy*** and a method called ***copy()***. Consider the below given program to understand the concept.

```
>>> class Point:
        pass

>>> p1=Point()
>>> p1.x=10
>>> p1.y=20

>>> import copy                  #import module copy
>>> p3=copy.copy(p1)             #use the method copy()
>>> print(p1)
     <__main__.Point object at 0x01581BF0>
>>> print(p3)
     <__main__.Point object at 0x02344A50>
>>> print(p3.x,p3.y)
     10 20
```

Observe that the physical address of the objects `p1` and `p3` are now different. But, values of attributes `x` and `y` are same. Now, use the following statements –

```
>>> p1 is p3
     False
>>> p1 == p3
     False
```

Here, the `is` operator gives the result as False for the obvious reason of `p1` and `p3` are being two different entities on the memory. But, why `==` operator is generating False as the result, though the contents of two objects are same? The reason is – `p1` and `p3` are the objects of user-defined type. And, Python cannot understand the meaning of equality on the new data type. The default behavior of equality (`==`) is identity (`is` operator) itself. Hence, Python applies this default behavior on `p1 == p3` and results in `False`.

(**NOTE:** If we need to define the meaning of equality (==) operator explicitly on user-defined data types (i.e. on class objects), then we need to override the method `___eq__()` inside the class. This will be discussed later in detail.)

The ***copy()*** method of ***copy*** module duplicates the object. The content (i.e. attributes) of one object is copied into another object as we have discussed till now. But, when an object itself is an attribute inside another object, the duplication will result in a strange manner. To understand this concept, try to copy Rectangle object (created in previous section) as given below –

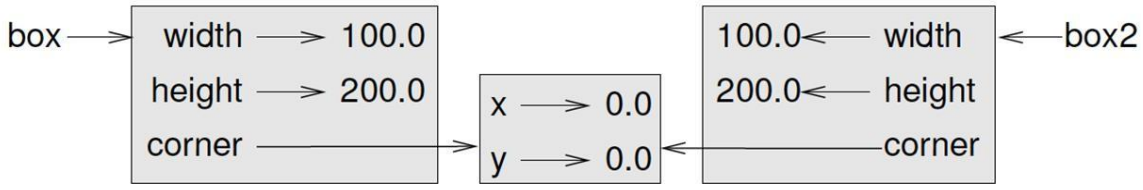
```
import copy
class Point:
    """ This is a class Point
        representing coordinate point
    """

class Rectangle:
    """ This is a class Rectangle.
    Attributes: width, height and Corner Point
    """

box1=Rectangle()
box1.corner=Point()
box1.width=100
box1.height=200
box1.corner.x=0
box1.corner.y=0

box2=copy.copy(box1)
print(box1 is box2)                    #prints False
print(box1.corner is box2.corner)      #prints True
```

Now, the question is – why **box1.corner** and **box2.corner** are same objects, when
**box1** and **box2** are different? Whenever the statement

```
box2=copy.copy(box1)
```

is executed, the contents of all the attributes of `box1` object are copied into the respective attributes of `box2` object. That is, `box1.width` is copied into `box2.width`, `box1.height` is copied into `box2.height`. Similarly, `box1.corner` is copied into `box2.corner`. Now, recollect the fact that corner is not exactly the object itself, but it is a reference to the object of type Point (Read the discussion done for Figure 4.1 at the beginning of this Chapter). Hence, the value of reference (that is, the physical address) stored in `box1.corner` is copied into `box2.corner`. Thus, the physical object to which `box1.corner` and `box2.corner` are pointing is only one. This type of copying the objects is known as ***shallow copy.*** To understand this behavior, observe the following diagram

Shallow Copy




Now, the attributes `width` and `height` for two objects `box1` and `box2` are independent. Whereas, the attribute `corner` is shared by both the objects. Thus, any modification done to `box1.corner` will reflect `box2.corner` as well. Obviously, we don't want this to happen, whenever we create duplicate objects. That is, we want two independent physical objects. Python provides a method ***deepcopy()*** for doing this task. This method copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on.

```
box3=copy.deepcopy(box1)
print(box1 is box3)                    #prints False
print(box1.corner is box3.corner)      #prints False
```

Thus, the objects `box1` and `box3` are now completely independent.

### 4.1.5 Debugging

While dealing with classes and objects, we may encounter different types of errors. For example, if we try to access an attribute which is not there for the object, we will get ***AttributeError***. For example –

```
>>> p= Point()
>>> p.x = 10
>>> p.y = 20
>>> print(p.z)
    AttributeError: 'Point' object has no attribute 'z'
```

To avoid such error, it is better to enclose such codes within try/except as given below –

```
try:
      z = p.x
except AttributeError:
      z = 0
```

When we are not sure, which type of object it is, then we can use ***type()*** as –

```
>>> type(box1)
      <class '__main__.Rectangle'>
```

Another method ***isinstance()*** helps to check whether an object is an instance of a particular class –

```
>>> isinstance(box1,Rectangle)
      True
```

When we are not sure whether an object has a particular attribute or not, use a function ***hasattr()***

```
>>> hasattr(box1, 'width')
 True
```

Observe the string notation for second argument of the function ***hasattr()***. Though the attribute `width` is basically numeric, while giving it as an argument to function ***hasattr()***, it must be enclosed within quotes.

## 4.2 CLASSES AND FUNCTIONS

Though Python is object oriented programming languages, it is possible to use it as functional programming. There are two types of functions viz. ***pure functions*** and ***modifiers***. A pure function takes objects as arguments and does some work without modifying any of the original argument. On the other hand, as the name suggests, modifier function modifies the original argument.

In practical applications, the development of a program will follow a technique called as ***prototype*** and ***patch***. That is, solution to a complex problem starts with simple prototype and incrementally dealing with the complications.

### 4.2.1 Pure Functions

To understand the concept of pure functions, let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes. Write a function to print time in HH:MM:SS format and another function to add two time objects. Note that, adding two time objects should yield proper result and hence we need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.

```
class Time:
    """Represents the time of a day
    Attributes: hour, minute, second """

def printTime(t):
    print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))

def add_time(t1,t2):
    sum=Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute +
    t2.minute sum.second =
    t1.second + t2.secondif
    sum.second >= 60: sum.second -=
    60 sum.minute += 1
     if sum.minute >=
           60:
        sum.minute -=
        60 sum.hour
        += 1

   return sum

t1=Time()
t1.hour=10
t1.minute=34
t1.second=25
print("Time1
is:")
printTime(t1)

t2=Time()
t2.hour=2
t2.minute=12
t2.second=41
print("Time2 is
:")
printTime(t2)
```

```
t3=add_time(t1,t2)
print("After adding two time objects:")
printTime(t3)
```

The output of this program would be – Time1 is: 10:34:25
Time2 is : 02:12:41
After adding two time objects: 12:47:06

Here, the function `add_time()` takes two arguments of type `Time`, and returns a `Time` object, whereas, it is not modifying contents of its arguments t1 and t2. Such functions are called as *pure functions.*

### 4.2.2 Modifiers

Sometimes, it is necessary to modify the underlying argument so as to reflect the caller. That is, arguments have to be modified inside a function and these modifications should be available to the caller. The functions that perform such modifications are known as ***modifier function.*** Assume that, we need to add few seconds to a time object, and get a new time. Then, we can write a function as below –

```
def increment(t, seconds):
    t.second += seconds

    while t.second >= 60:
        t.second -= 60
        t.minute += 1

    while t.minute >= 60:
        t.minute -= 60
        t.hour += 1
```

In this function, we will initially add the argument `seconds` to `t.second`. Now, there is a chance that `t.second` is exceeding 60. So, we will increment minute counter till `t.second` becomes lesser than 60. Similarly, till the `t.minute` becomes lesser than 60, we will decrement minute counter. Note that, the modification is done on the argument `t` itself. Thus, the above function is a ***modifier.***

### 4.2.3 Prototyping v/s Planning

Whenever we do not know the complete problem statement, we may write the program initially, and then keep of modifying it as and when requirement (problem definition) changes. This methodology is known as ***prototype and patch.*** That is, first design the prototype based on the information available and then perform patch-work as and when extra information is gathered. But, this type of incremental development may end-up in unnecessary code, with many special cases and it may be unreliable too.

An alternative is ***designed development***, in which high-level insight into the problem can make the programming much easier. For example, if we consider the problem of adding two time objects, adding seconds to time object etc. as a problem involving numbers with base 60 (as every hour is 60 minutes and every minute is 60 seconds), then our code can be improved. Such improved versions are discussed later in this chapter.

### 4.2.4 Debugging

In the program written in Section 4.2.1, we have treated time objects as valid values. But, what if the attributes (second, minute, hour) of time object are given as wrong values like negative number, or hours with value more than 24, minutes/seconds with more than 60 etc? So, it is better to write error-conditions in such situations to verify the input. We can write a function similar to as given below –

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False

    if time.minute >= 60 or time.second >= 60:
        return False

    return True
```

Now, at the beginning of `add_time()` function, we can put a condition as –

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')

    #remaining statements of add_time() functions
```

Python provides another debugging statement ***assert***. When this keyword is used, Python evaluates the statement following it. If the statement is True, further statements will be evaluated sequentially. But, if the statement is False, then ***AssertionError*** exception is raised. The usage of *assert* is shown here –

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    #remaining statements of add_time() functions
```

The *assert* statement clearly distinguishes the normal conditional statements as a part of the logic of the program and the code that checks for errors.

## 4.3 CLASSES AND METHODS

The classes that have been considered till now were just empty classes without having any definition. But, in a true object oriented programming, a class contains class-level attributes, instance-level attributes, methods etc. There will be a tight relationship between the object of the class and the function that operate on those objects. Hence, the object oriented nature of Python classes will be discussed here.

### 4.3.1 Object-Oriented Features

As an object oriented programming language, Python possess following characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact.

To establish relationship between the object of the class and a function, we must define a function as a member of the class. A function which is associated with a particular class is known as a ***method***. Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

Now onwards, we will discuss about classes and methods.

### 4.3.2 The __init__() Method

(A method__init__() has to be written with two underscores before and after the word *init*)

Python provides a special method called as `__init__()` which is similar to constructor method in other programming languages like C++/Java. The term *init* indicates initialization. As the name suggests, this method is invoked automatically when the object of a class is created. Consider the example given here –

```
import math

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

    def dist(self,p2):
        d=math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)
        return d

    def __str__(self):
        return "(%d,%d)"%(self.x, self.y)
```

```
p1=Point(10,20)          #__init__() is called automatically
p2=Point(4,5)            #__init__() is called automatically

print("P1 is:",p1)       #__str__() is called automatically
print("P2 is:",p2)       #__str__() is called automatically

d=p1.dist(p2)            #explicit call for dist()

print("The distance is:",d)
```

The sample output is –

```
P1 is: (10,20)
P2 is: (4,5)
Distance is: 16.15549442140351
```

Let us understand the working of this program and the concepts involved:

- Keep in mind that every method of any class must have the first argument as ***self***. The argument ***self*** is a reference to the current object. That is, it is reference to the object which invoked the method. (Those who know C++, can relate ***self*** with ***this*** pointer). The object which invokes a method is also known as ***subject***.

- The method `__init__()` inside the class is an initialization method, which will be invoked automatically when the object gets created. When the statement like

  ```
  p1=Point(10,20)
  ```

  is used, the `__init__()` method will be called automatically. The internal meaning of the above line is –

  ```
  p1.__init__(10,20)
  ```

  Here, p1 is the object which is invoking a method. Hence, reference to this object is created and passed to `__init__()` as *`self`*. The values 10 and 20 are passed to formal parameters a and b of `__init__()` method. Now, inside `__init__()` method, we have statements

  ```
  self.x=10
  self.y=20
  ```

  This indicates, `x` and `y` are instance attributes. The value of `x` for the object `p1` is 10 and, the value of `y` for the object `p1` is 20.

  When we create another object `p2`, it will have its own set of `x` and `y`. That is, memory locations of instance attributes are different for every object.

  ***Thus, state of the object can be understood by instance attributes.***

- The method `dist()` is an ordinary member method of the class `Point`. As mentioned earlier, its first argument must be self. Thus, when we make a call as –

```
d=p1.dist(p2)
```

a reference to the object `p1` is passed as `self` to `dist()` method and `p2` is passed explicitly as a second argument. Now, inside the `dist()` method, we are calculating distance between two point (Euclidian distance formula is used) objects. Note that, in this method, we cannot use the name `p1`, instead we will use `self` which is a reference (alias) to `p1`.

- The next method inside the class is ***__str__(). It is a special method used for string representation of user-defined object.*** Usually, `print()` is used for printing basic types in Python. But, user-defined types (class objects) have their own meaning and a way of representation. To display such types, we can write functions or methods like `print_point()` as we did in Section 4.1.2. But, more polymorphic way is to use `__str__()` so that, when we write just `print()` in the main part of the program, the `__str__()` method will be invoked automatically. Thus, when we use the statement like –

```
print("P1 is:",p1)
```

the ordinary print() method will print the portion "P1 is:" and the remaining portion is taken care by `__str__()` method. In fact, `__str_()` method will return the string format what we have given inside it, and that string will be printed by `print()` method.

### 4.3.3 Operator Overloading

***Ability of an existing operator to work on user-defined data type (class)*** is known as operator overloading. It is a polymorphic nature of any object oriented programming. Basic operators like +, -, * etc. can be overloaded. To overload an operator, one needs to write a method within user-defined class. Python provides a special set of methods which have to be used for overloading required operator. The method should consist of the code what the programmer is willing to do with the operator. Following table shows gives a list of operators and their respective Python methods for overloading.

| Operator | Special Function in Python | Operator | Special Function in Python |
|---|---|---|---|
| + | __add__() | <= | __le__() |
| - | __sub__() | >= | __ge__() |
| * | __mul__() | == | __eq__() |

| | | | |
|---|---|---|---|
| / | __truediv__() | != | __ne__() |
| % | __mod__() | in | __contains__() |
| < | __lt__() | len | __len__() |
| > | __gt__() | str | __str__() |

Let us consider an example of Point class considered earlier. Using operator overloading, we can try to add two point objects. Consider the program given below –

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

    def __add__(self, p2):
        p3=Point()
        p3.x=self.x+p2.x
        p3.y=self.y+p2.y
        return p3

    def __str__(self):
        return "(%d,%d)"%(self.x, self.y)
p1=Point(10,20)
p2=Point(4,5)
print("P1 is:",p1)
print("P2 is:",p2)
p4=p1+p2                    #call for __add__() method
print("Sum is:",p4)
```

The output would be –

```
P1 is: (10,20)
P2 is: (4,5)
Sum is: (14,25)
```

In the above program, when the statement `p4 = p1+p2` is used, it invokes a special method `__add__()` written inside the class. Because, internal meaning of this statement is–

```
p4 = p1.__add__(p4)
```

Here, `p1` is the object invoking the method. Hence, `self` inside `__add__()` is the reference (alias) of `p1`. And, `p4` is passed as argument explicitly.

In the definition of__add__(), we are creating an object p3 with the statement –

```
p3=Point()
```

The object p3 is created without initialization. Whenever we need to create an object with and without initialization in the same program, we must set arguments of_init__() for some default values. Hence, in the above program arguments a and b of_init__() are made as default arguments with values as zero. Thus, x and y attributes of p3 will be now zero. In the_add_() method, we are adding respective attributes of self and p2 and storing in p3.x and p3.y. Then the object p3 is returned. This returned object is received as p4 and is printed.

**NOTE** that, in a program containing operator overloading, the overloaded operator behaves in a normal way when basic types are given. That is, in the above program, if we use the statements

```
m= 3+4
print(m)
```

it will be usual addition and gives the result as 7. But, when user-defined types are used as operands, then the overloaded method is invoked.

Let us consider a more complicated program involving overloading. Consider a problem of creating a class called Time, adding two Time objects, adding a number to Time object etc. that we had considered in previous section. Here is a complete program with more of OOP concepts.

```
class Time:
    def__init__(self, h=0,m=0,s=0):
        self.hour=h
        self.min=m
        self.sec=s

    def time_to_int(self):
        minute=self.hour*60+self.min
        seconds=minute*60+self.sec
        return seconds

    def int_to_time(self, seconds):
        t=Time()
        minutes, t.sec=divmod(seconds,60)
        t.hour, t.min=divmod(minutes,60)
        return t

    def___str__(self):
        return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)
```

```
    def __eq__(self,t):
        return self.hour==t.hour and self.min==t.min and self.sec==t.sec

    def __add__(self,t):
        if isinstance(t, Time):
            return self.addTime(t)
        else:
            return self.increment(t)

    def addTime(self, t):
        seconds=self.time_to_int()+t.time_to_int()
        return self.int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return self.int_to_time(seconds)

    def __radd__(self,t):
        return self.__add__(t)

T1=Time(3,40)
T2=Time(5,45)
print("T1 is:",T1)
print("T2 is:",T2)
print("Whether T1 is same as T2?",T1==T2)      #call for__eq__()

T3=T1+T2                        #call for__add__()

print("T1+T2 is:",T3)

T4=T1+75                        #call for__add__()
print("T1+75=",T4)

T5=130+T1                       #call for__radd__()
print("130+T1=",T5)

T6=sum([T1,T2,T3,T4])
print("Using sum([T1,T2,T3,T4]):",T6)
```

The output would be –

```
T1 is: 03:40:00
T2 is: 05:45:00
Whether T1 is same as T2? False
T1+T2 is: 09:25:00
T1+75= 03:41:15
130+T1= 03:42:10
Using sum([T1,T2,T3,T4]): 22:31:15
```

Working of above program is explained here

- The class Time has `__init__()` method for initialization of instance attributes `hour`, `min` and `sec`. The default values of all these are being zero.
- The method `time_to_int()` is used convert a Time object (hours, min and sec) into single integer representing time in number of seconds.
- The method `int_to_time()` is written to convert the argument `seconds` into time object in the form of hours, min and sec. The built-in method ***divmod()*** gives the quotient as well as remainder after dividing first argument by second argument given to it.
- Special method `__eq__()` is for overloading equality (==) operator. We can say one Time object is equal to the other Time object if underlying hours, minutes and seconds are equal respectively. Thus, we are comparing these instance attributes individually and returning either True of False.
- When we try to perform addition, there are 3 cases –
  - Adding two time objects like `T3=T1+T2`.
  - Adding integer to Time object like `T4=T1+75`
  - Adding Time object to an integer like `T5=130+T1`

  Each of these cases requires different logic. When first two cases are considered, the first argument will be `T1` and hence `self` will be created and passed to `__add__()` method. Inside this method, we will check the type of second argument using `isinstance()` method. If the second argument is Time object, then we call `addTime()` method. In this method, we will first convert both Time objects to integer (seconds) and then the resulting sum into Time object again. So, we make use `time_to_int()` and `int_to_time()` here. When the 2nd argument is an integer,

  it is obvious that it is number of seconds. Hence, we need to call `increment()` method.
  Thus, based on the type of argument received in a method, we take appropriate action. This is known as ***type-based dispatch.***

  In the 3rd case like `T5=130+T1`, Python tries to convert first argument 130 into `self`, which is not possible. Hence, there will be an error. This indicates that for Python, `T1+5` is not same as `5+T1` (Commutative law doesn't hold good!!). To avoid the possible error, we need to implement ***right-side addition*** method `__radd__()`. Inside this method, we can call overloaded method `__add__()`.
- The beauty of Python lies in surprising the programmer with more facilities!! As we have implemented `__add__()` method (that is, overloading of + operator), the built-in `sum()` will is capable of adding multiple objects given in a sequence. This is due to ***Polymorphism*** in Python***.*** Consider a list containing Time objects, and then call `sum()` on that list as –

```
T6=sum([T1,T2,T3,T4])
```

The `sum()` internally calls `__add__()` method multiple times and hence gives the appropriate result. Note down the square-brackets used to combine Time objects as a list and then passing it to `sum()`.

Thus, the program given here depicts many features of OOP concepts.

## Type-based dispatch-

It dispatches the computation to different methods based on the type of the arguments.

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

 def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

### 4.3.4 Debugging

We have seen earlier that ***hasattr()*** method can be used to check whether an object has particular attribute. There is one more way of doing it using a method ***vars()***. This method maps attribute names and their values as a dictionary. For example, for the Point class defined earlier, use the statements –

```
>>> p = Point(3, 4)
>>> vars(p)                #output is {'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
     for attr in vars(obj):
          print(attr, getattr(obj, attr))
```

Here, `print_attributes()` traverses the dictionary and prints each attribute name and its corresponding value. The built-in function `getattr()` takes an object and an attribute name (as a string) and returns the attribute's value.

# GENERAL OOP CONCEPTS

At the earlier age of computers, the programming was done using assembly language. Even though, the assembly language can be used to produce highly efficient programs, it is not easy to learn or to use effectively. Moreover, debugging assembly code is quite difficult. At the later stage, the programming languages like BASIC, COBOL and FORTRAN came into existence. But, these languages are non-structured and consisting of a mass of tangled jumps and conditional branches that make a program virtually impossible to understand.

To overcome these problems, the structured or procedural programming methodology was developed. Here, the actual problem is divided into small independent tasks/modules. Then the programs are written for each of these tasks and they are grouped together to get the final solution for the given problem. Thus, the solution design technique for this method is known as *top-down approach*. C is the one successful language that adopted structured programming style. However, even with the structured programming methods, once a project/program reaches a certain size, its complexity exceeds what a programmer can manage.

The new approach – object oriented programming was developed to overcome the problems with structured approach. In this methodology, the actual data and the operations to be performed on that are grouped as a single entity called object. The objects necessary to get the solution of a problem are identified initially. The interactions between various objects are then identified to achieve the solution of original problem. Thus, it is also known as *bottom-up approach.* Object oriented concepts inhabits many advantages like re-usability of the code, security etc.

In structured programming approach, the programs are written around *what is happening* rather than *who is being affected.* That is, structured programming focuses more on the process or operation and not on the data to be processed. This is known as *process oriented model* and this can be thought of as *code acting on data.* For example, a program written in C is defined by its functions, any of which may operate on any type of data used by the program.

But, the real world problems are not organized into data and functions independent of each other and are tightly closed. So, it is better to write a program around '*who is being affected'.* This kind of data-centered programming methodology is known as *object oriented programming* (*OOP*) and this can be thought of as *data controlling access to code.* Here, the behavior and/or characteristics of the data/objects are used to determine the function to be written for applying them. Thus, the basic idea behind OOP language is to combine both data and the function that operates on data into a single unit. Such a unit is called as an *object*. A function that operates on data is known as a *member function* and it is the only means of accessing an object's data.

**Elements of OOP:** The object oriented programming supports some of the basic concepts as building blocks. Every OOPs language normally supports and developed around these features. They are discussed hereunder.

**Class: A class is a user defined data type which binds data and functions together into a single entity.**

Class is a building block of any object oriented language. As it is discussed earlier, object oriented programming treats data and the code acting on that data as a connected component. That is, data and code are not treated separately as procedure oriented languages do. Thus, OOPs suggests to wrap up the data and functions together into a single entity. Normally, a class represents the prototype of a real world entity. Hence, a class, by its own, is not having any physical existence. It can be treated as a user-defined data type.

Since a class is a prototype or blueprint of a real world entity, it consists of number of properties (known as data members) and behavior (known as member functions). To illustrate this, consider an example of a class representing a human being shown in the following Figure –

| **Human Being** |
|---|
| - Hair Color<br>- Number of legs<br>- Number of eyes<br>- Skin Color<br>- Gender |
| + Walking<br>+ Talking<br>+ Eating |

**Class diagram for Human Being**

Few of the properties of human can be *number of legs, number of eyes, gender, number of hands, hair color, skin color etc.* And the functionality or behavior of a human may be *walking, talking, eating, thinking* etc.

**Object : An object is an instance of a class.**

A class is just a prototype, representing a new data type. To use this new data type, we need to create a variable of this type. Such a variable is known as an object. Thus, an object is a physical entity, which consumes memory. In OOPs, the object represents a real world data which defines the properties and behavior of any real world entity. Every object has its unique existence and it is different from the other objects of a same class.

Let us refer to the class of human being discussed in previous section. Assume that there are two persons: AAA and BBB. Now, properties of AAA and BBB may be having different values as shown in the following Table.

**Properties of Objects**

| Property/Attribute | Objects | |
|---|---|---|
| | AAA | BBB |
| Skin color | Whitish | Fair |
| Hair | gray | black |
| Number of legs | 2 | 2 |
| Number of eyes | 2 | 2 |

Also, the walking and talking style of both of them may be different. Hence there are two different objects of the same class.

Thus, two or more objects of the same class will differ in the values of their properties and way they behave. But, they share common set of types of properties and behavior.

**Encapsulation: The process of binding data and code together into a single entity is called encapsulation.**

It is the mechanism that binds code and the data it manipulates together and keeps both safe from misuse and unauthorized manipulation. In any OOP language, the basis of encapsulation is the *class.* Class defines the structure and behavior (i.e. data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. So, object is also referred to as an *instance of a class* and it is thought just as a variable of user-defined data type. Thus, class is a logical construct and an object is a physical entity. The data defined by the class are known as *member variables* and the functions written to operate on that data are known as *member functions or methods.* The member variables and functions can be *private* or *public.* If a particular member is private, then only the other members of that class can access that member. That is, a private member of the class can't be accessed from outside the class. But, if any member is public, then it can be accessed from anywhere in the program. Thus, through encapsulation, an OOP technique provides high security for user's data and for the entire system.

**Data Abstraction: Hiding the implementation details from the end user.**

Many a times, abstraction and encapsulation are used interchangeably. But, actually, they are not same. In OOPs, the end user of the program need not know how the actual function works, or what is being done inside a function to make it work. The user must know only the abstract meaning of the task, and he can freely call a function to do that task. The internal details of function may not be known to the user. Designing the program in such a way is called as abstraction.

To understand the concept of abstraction, consider a scenario: When you are using Microsoft Word, if you click on Save icon, a dialogue box appears which allows you to save the document in a physical location of your choice. Similarly, when you click Open icon, another dialogue box appears to select a file to be opened from the hard disk. You, as a

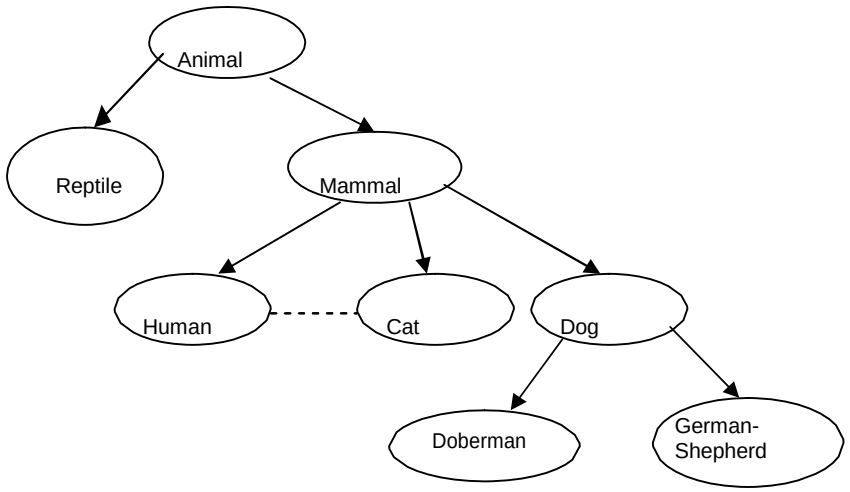
user will not be knowing how the internal code would have written so as to open a dialogue box when an icon is being clicked. As a user, those details are not necessary for you. Thus, such implementation details are hidden from the end user. This is an abstraction.

Being an OOPs programmer, one should design a class (with data members and member functions) such a way that, the internal code details are hidden from the end user. OOPs provide a facility of having member functions to achieve this technique and the external world (normally, a main() function) needs to call the member function using an object to achieve a particular task.

**Inheritance: Making use of existing code.**

It is a process by which one object can acquire the properties of another object. It supports the concept of hierarchical (top-down) classification. For example, consider a large set of animals having their own behaviors. In that, mammals are of one kind having all the properties of animals with some additional behaviors that are unique to them.

We can divide mammals class into various mammals like dogs, cats, human etc. Again among the dogs, differentiation is there like Doberman, German-shepherd, and Labrador etc. Thus, if we consider a German-shepherd, it is having all the qualities of a dog along with its own special features. Moreover, it exhibits all the properties of a mammal, and in turn of an animal. Hence it is inheriting the properties of animals, then of mammals and then of dogs along with its own specialties.



'Normally, inheritance of this type is also known as "is-a" relationship. Because, we can easily say "Doberman ***is a*** dog", "Dog ***is a*** mammal" etc. Hence, inheritance is termed as ***Generalization to Specialization*** if we consider from top-to-bottom level. On the other hands, it can be treated as ***Specialization to Generalization*** if it is bottom-to-top level.

This indicates, in inheritance, the topmost base class will be more generalized with only properties which are common to all of its derived classes (various levels) and the bottom-most class is most specialized version of the class which is ready to use in a real-world.

If we apply this concept for programming, it can be easily understood that a code written is reusable. Thus, in this mechanism, it is possible for one object to be a specific instance of a more general case. Using inheritance, an object need only define those qualities that make it unique object within its class. It can inherit its general attributes from its parent.

**Polymorphism:** ***This can be thought of as one interface, multiple methods.***
It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Using this mechanism, function overloading and operator overloading can be done.

Consider an example of performing stack operation on three different types of data viz. integer, floating-point and characters. In a non-object oriented programming, we have to write functions with different name for push and pop operations for all these types of data even though the logic is same for all the data types. But in OOP languages, we can use the same function names with the data types of the parameters being different. This is an example for function overloading.We know that the '+' operator is used for adding two numbers. Conceptually, the concatenation of two strings is also an addition. But, in non-object oriented programming language, we cannot use '+' operator to concatenate two strings. This is possible in object oriented programming language by overloading the '+' operator for string operands.

Polymorphism is also meant for *economy of expression.* That is, the way you express things is more economical when you use polymorphism. For example, if you have a function to add two matrices, you can use just a + symbol as:

m3 = m1 + m2;

here, m1, m2 and m3 are objects of matrix class and + is an overloaded operator. In the same program, if you have a function to concatenate two strings, you can write an overloaded function for + operator to do so –

s3= s1+ s2; where s1, s2 and s3 are strings.

Moreover, for adding two numbers, the same + operator is used with its default behavior. Thus, one single operator + is being used for multiple purposes without disturbing its abstract meaning – addition. But, type of data it is adding is different. Hence, the way you have expressed your statements is more economical rather than having multiple functions like –

addMat(m1, m2); concat(s1, s2); etc.

```
class Person:
  def_init_(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
x = Person("Guido", "Van")
x.printname()
```

To create a class that inherits the functionality from another class, send the parent class.as a parameter when creating the child class.

```
class Student(Person):
  pass
x = Student("Charles", "Severance")
x.printname()
```

```
class Student(Person):
  def_init_(self, fname, lname):
    #add properties etc.
```

- The child's__init__() function overrides the inheritance of the parent's__init__() function.
- To keep the inheritance of the parent's__init__() function, add a call to the parent's __init__() function

```
class Student(Person):
  def_init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Python also has a super() function that will make the child class inherit all the methods and properties from its parent.

```
class Student(Person):
  def_init_(self, fname, lname):
    super().__init__(fname, lname)
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

```
class Student(Person):
  def__init__(self, fname, lname, year):
    super()._init_(fname, lname)
    self.graduationyear = year

  def  greeting(self):
    print(“We welcome", self.firstname, self.lastname, "to the class
of", self.graduationyear)

x = Student(“ABC", “BBB", 2020)
x.greeting()
```

**Inheritance Example:**

```
class Person:
  def_init_(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
x = Person("Guido", "Van")
x.printname()

class Student(Person):
  def__init__(self, fname, lname):
      Person.__init__(self, fname, lname)

x = Student("Charles", "Severance")
x.printname()
```

**o/p**
**Guido Van**
**Charles Severance**

**#usage of super function**

```
class Person:
  def_init_(self, fname, lname):
    self.firstname = fname
    self.lastname = lname
  def printname(self):
    print(self.firstname, self.lastname)
```

```
class Student(Person):
  def__init__(self, fname, lname, year):
    super()._init_(fname, lname)
    self.graduationyear = year
  def  greeting(self):
    print("We welcome", self.firstname, self.lastname, "to the class
of", self.graduationyear)

x = Student("ABC", "BBB", 2020)
x.greeting()
```

**o/p**

**We welcome ABC BBB to the class of 2020**

## Access Modifiers in Python

The access modifiers in Python are used to modify the default scope of variables. There are three types of access modifiers in Python: public, private, and protected.

Variables with the public access modifiers can be accessed anywhere inside or outside the class, the private variables can only be accessed inside the class, while protected variables can be accessed within the same package.

To create a private variable, you need to prefix double underscores with the name of the variable. To create a protected variable, you need to prefix a single underscore with the variable name. For public variables, you do not have to add any prefixes at all.

```
class Car:
    def___init__(self):
        print ("Car Description")
        self.name = "Ritz"
        self._maker = "Maruthi Suzuki"
        self._model = 2000

    def usepri(self):
        print(self.__maker)



c=Car()
print(c._model)
print(c.name)
#print(c.__maker)
#  the  above  statement  –  private  attribute  accesses  results  in
#AttributeError
c.usepri()#to access private attribute use class method
```

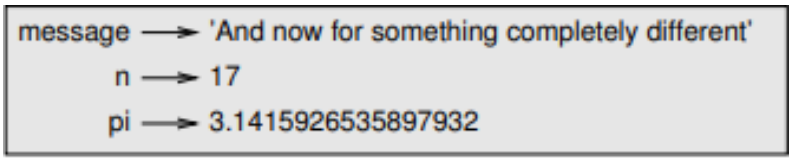
```
class Car:
    def __init__(self):
        print ("Car Description")
        self.name = "Ritz"
        self._maker = "Maruthi Suzuki"
        self._model = 2000
    def usepri(self):
        print(self.__maker)



c=Car()

class Bike(Car):
    pass

b=Bike()
print(b._model)
print(b.__maker)#results in Attribute error
```

## Access Modifiers-Methods

```
class Base:
    def show(self):
        print("public method")
        Base.__display(self)
    def_display(self):
        print("private method")
    def _printval(self):
        print("protected")
        Base.__display(self)


b=Base()
b.show()
b._printval()

class Derive(Base):
    pass

d=Derive()
d._printval()
d.show()
d.__display()
#The above statement execution results in AttributeError because
access to private method of base class
```

**State diagram**

A common way to represent variables on paper is to write the name with an arrow pointing to its value. This kind of figure is called a state diagram which shows the state of each variables.

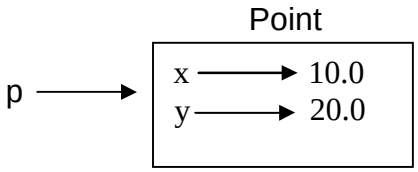



State diagram.

The stack diagram show the state of a program, and object diagram show the attributes of an object and their values.

**object diagram**

A state diagram that shows an object and its attributes is called as ***object diagram.*** For the object p, the object diagram is shown in Figure below.




Object Diagram

**Stack Diagram**

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```
def cat_twice(part1, part2):
cat = part1 + part2
print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception:

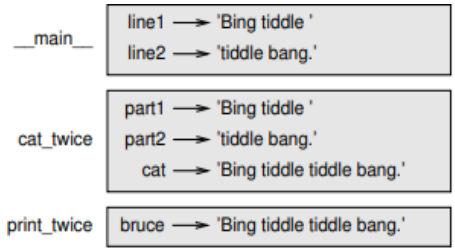
```
>>> print(cat)
NameError: name 'cat' is not defined
```

```
def print_twice(bruce):
print(bruce)
print(bruce)
```

Parameters are also local. For example, outside print_twice, there is no such thing as bruce.

**Stack diagrams**

To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.




Stack diagram.

**Stack Diagram:** A graphical representation of a stack of functions, their variables, and the values to which they refer.

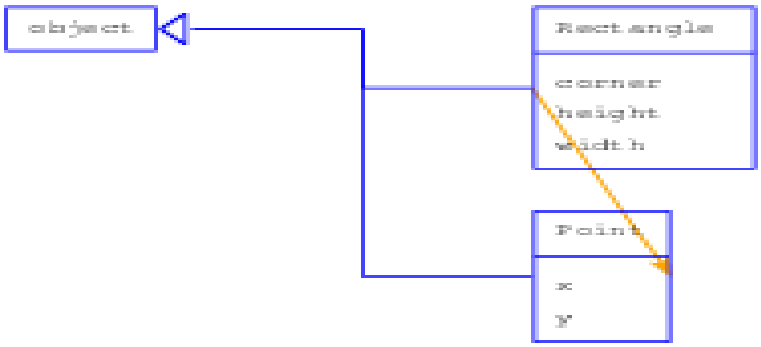
**Frame :** A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it.The frames are arranged in a stack that indicates which function called which, and so on. In this example, print_twice was called by cat_twice, and cat_twice was called by__main___, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to_main___

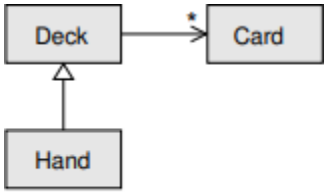
**Class diagrams**

A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them. There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point.This kind of relationship is called HAS-A, as in, “a Rectangle has a Point.”
- One class might inherit from another. This relationship is called IS-A
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a dependency. class diagram is a graphical representation of these relationship




Each class is represented with a box that contains the name of the class, any methods the class provides, any class variables, and any instance variables. In this example, Rectangle and Point have instance variables, but no methods or class variables. The arrow from Rectangle to Point shows that Rectangles contain an embedded Point. In addition, Rectangle and Point both inherit from object, which is represented in the diagram with a triangle-headed arrow.




Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck. The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has.

A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards. There are no dependencies in this diagram. They would normally be shown with a dashed arrow.