UNIT 4 OBJECT-ORIENTED CONCEPTS USED IN PYTHON

Features of object-oriented programming-Fundamental concepts- Class- Encapsulation- Inheritance- Polymorphism.
Object references - Turtle graphics - creating a Turtle Graphics Window - the "Default" Turtle - Fundamental Turtle Attributes and Behavior - Additional Turtle Attributes - Creating Multiple Turtles.

4.1 Features of object-oriented programming

- **Definition: Object-oriented programming** (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**.
- **Object Oriented** means directed towards objects.
- Python is an Object Oriented programming (OOP).
- It is a way of programming that focuses on using objects and classes to design and build applications.
- It is used to design the program using classes and objects.
- **Advantages of oops**:
    - It is faster
    - It is easy to execute
    - It provides a clear structure for the programs
    - Easy to maintain, modify and debug
    - It is used to create full reusable applications with less code and shorter development time

- **Features of oops:**
    - Class
    - Object
    - Encapsulation
    - Abstraction
    - Inheritance
    - Polymorphism

Class

- The class can be defined as a collection of objects.
- It is a logical entity that has some specific attributes and methods.
- A class is a blueprint for the object.
- A class is a template for objects
- A Class in Python is a logical grouping of data and functions.
- A class is a collection of objects

Syntax :

```
class classname:
    Class body
```

Object

- An object is an instance of a class.
- The **objector instance** contains real data or information.
- The object is an entity that has state and behaviour.
- Object as collection of both data and functions that operate on that data.
- An object is used to allocate the memory.
- Each object has own set of data members and member functions.

Syntax:

```
Objectname=classname()
```

Encapsulation

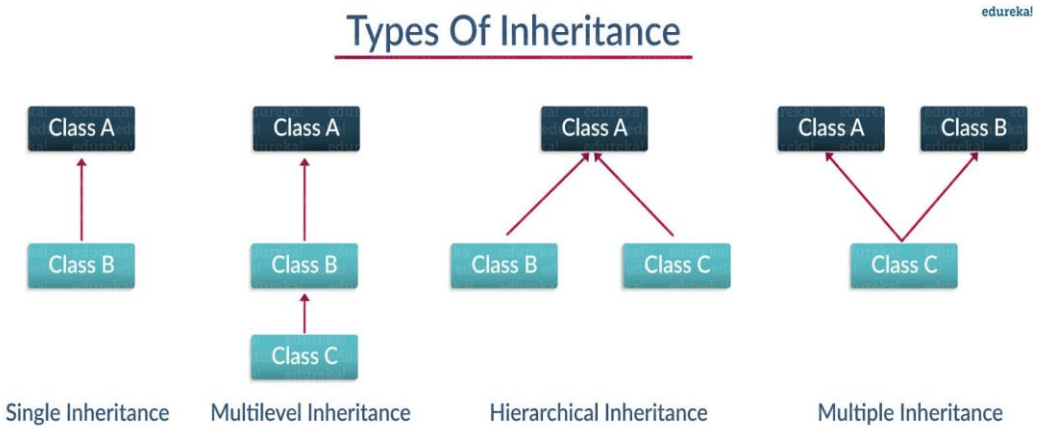
- Wrapping up of data and method into a single unit is called Encapsulation.
- It is used to restrict access to methods and variables.
- **Encapsulation** is a means of bundling together instance variables and methods to form a given type (class).
- Selected members of a class can be made inaccessible ("hidden") from its clients, referred to as *information hiding* .
- Information hiding is a form of abstraction.

Abstraction

- Abstraction is used to hide internal details and show only functionalities.
- It refers to essential information without including the background details.

Inheritance

- Deriving a new class from the old class is called inheritance.
- Old class is called parent class or super class or base class.
- New class is called child class or sub class or derived class.
- Reusability of coding is the main advantages of inheritance.



Polymorphism

- Poly means many and morph means forms.
- It means more than one form with the same name
- It means one task can be performed in different ways.
- There are types of polymorphism
    - Compile time polymorphism or Static polymorphism
    - Run time polymorphism or dynamic polymorphism
- Method is invoked at compile time is called compile time polymorphism. Ex. Method overloading
- Method is invoked at runtime is called run time polymorphism. Ex. Method overriding

4.2 Fundamental concepts

4.2.1 Class

- **Class Definition**: A **class** specifies the set of instance variables and methods that are "bundled together" for defining a type of object.
- Python class is a blueprint of an object.
- Class is a keyword

Syntax:

```
Class classname:
        Variables and functions
```

- **Object Definition:** An object is simply a collection of data (variables) and methods (functions) that act on those data.
- An object is also called an instance of a class

Syntax:

```
Objectname=classname()
```

- Call the variable and function in a class using the following

```
Objectname.variablename
Objectname.functionname()
```

Example:	**Output:**

class ruff:
  def f1(self):
  print("Hello World")
ob=ruff()
ob.f1()
	Hello World

- **Self definition:** The self parameter is a reference to the current instance of the class. It has to be the first parameter of any function in the class. It contains a reference to the object instance to which the method belongs.
- **Constructor Definition**:
  - Constructor is to initialize (assign values) to the data members of the class when an object of class is created.
  - In Python the __init__() method is called the constructor and is always called when an object is created.
  - Instance variables are initialized in the __init__ () method.

```
Syntax:
        def  __init__(self):
            # body of the constructor
```

Example:	**Output:**

class ruff:
  def __init__(self,a,b):
  self.a=a
  self.b=b
  def f(self):
  print(self.a,self.b)
ob=ruff(10,20)
ob.f()
	10 20

  - There are two types of constructor:
    1. **default constructor**

2. **parameterized constructor**

- **default constructor :**The default constructor is simple constructor which doesn't accept any arguments.

Example:	**Output:**

class ruff:
  def __init__(self):
  print("Hello")
ob=ruff()
	Hello

- **parameterized constructor :**constructor with parameters is known as parameterized constructor. First argument is self and the rest of the arguments are provided by the programmer.

Example:	**Output:**

class ruff:
  def __init__(self,a,b):
  self.a=a
  self.b=b
  print(self.a,self.b)
 	
ob=ruff(10,20)
	10 20

- **del** keyword is used to delete an object.
- delete properties on objects by using the **del** keyword

Syntax:
del objectname del objectname.variablename

4.2.2 Encapsulation

- **Encapsulation** is a means of bundling together instance variables and methods to form a given type (class).
- Selected members of a class can be made inaccessible ("hidden") from its clients, referred to as **information hiding** .
- Information hiding is a form of **abstraction**.
- Private members of a class begin with two underscore characters, and cannot be directly accessed.

Example:	**Output:**

class ruff:
def __init__(self,x,y):
  self.__a=x
  self.__b=y
	30 20
	
	AttributeError: 'ruff' object has no

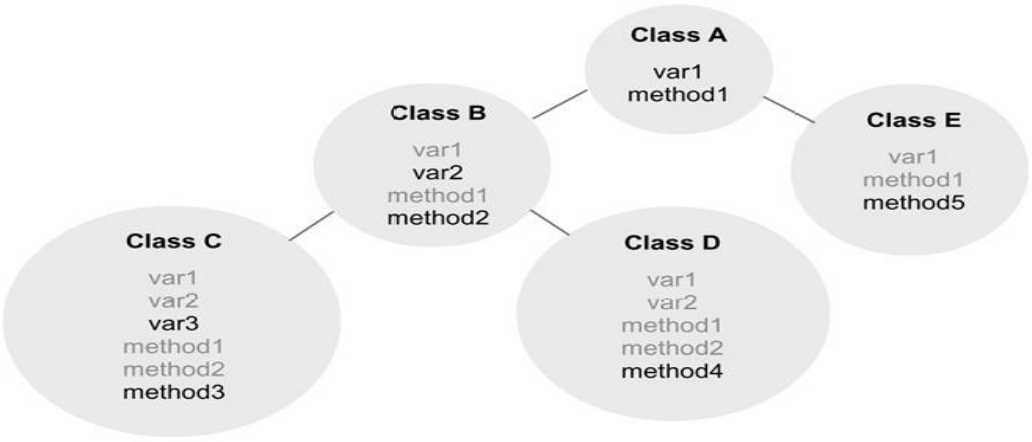
print(self.__a,self.__b) ob=ruff(30,20) print(ob.__a)	attribute '__a'

- In the above example __a and __b are private variables and cannot be accessed directly.
- Renaming of identifiers is called ***name mangling*** .
- **Special methods in Python:**
  - Special methods in Python have names that begin and end with two underscore characters, and are automatically called in Python.
  - __init__() - it is automatically called whenever a new object is created.
  - __str__ () - it is called when an object is displayed using print.
  - __repr__() – it is called when the value of an object is displayed in the Python shell .

Methods	**Meaning**
a.__init__(self, args)	constructor: a = A(args)
a.__del__(self)	destructor: del a
a.__str__(self)	pretty print: print a, str(a)
a.__repr__(self)	representation: a = eval(repr(a))
a.__add__(self, b)	a + b
a.__sub__(self, b)	a - b
a.__mul__(self, b)	a*b
a.__div__(self, b)	a/b
a.__lt__(self, b)	a < b
a.__gt__(self, b)	a > b
a.__le__(self, b)	a <= b
a.__ge__(self, b)	a => b
a.__eq__(self, b)	a == b
a.__ne__(self, b)	a != b

4.2.3 Inheritance

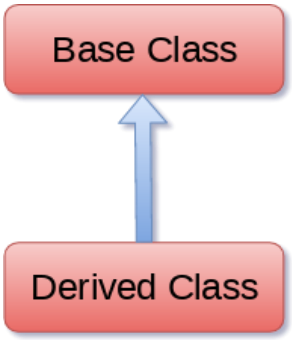
- **Inheritance** is the ability of a class to inherit members of another class as part of its own definition.
- The inheriting class is called a **subclass** (also “derived class” or “child class”), and the class inherited from is called the **superclass** (also “base class” or “parent class”).
- Class hierarchy is as follows:



- Class A is a super class. Classes B & E are subclasses of class A, both are inherited variables and methods of class A. Class C & D are direct subclasses of class B but indirect subclasses of class A.
- **Definition of super() : super()** function that will make the child class inherit all the methods and properties from its parent
- **Types of inheritance:**
    1. Single inheritance
    2. Multilevel inheritance
    3. Multiple inheritance
    4. Hierarchical inheritance
    5. Hybrid inheritance

1. Single inheritance

- Only one child class inherit only one parent class is called single inheritance.



Syntax:

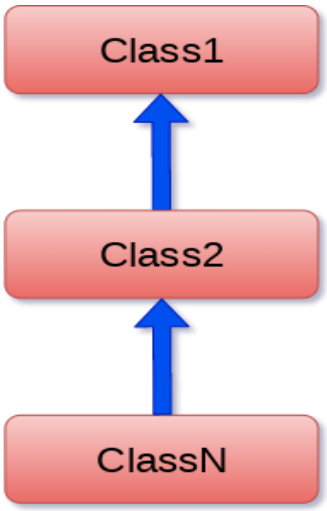
```
class parent:
        Statement
class child(parent):
        Statement
```

Example :	**Output:**

class one:
  def f1(self):
  print("parent class")
class two(one):
  def f2(self):
  print("child class")
 	
ob=two()
ob.f1()
ob.f2()
	parent class
	child class

2. Multilevel inheritance

- Multi-level inheritance is archived when a derived class inherits another derived class. .



Syntax:

class parent:
  Statement
class child1(parent):
  Statement
class child2(child1):
  Statement

Example :	**Output:**

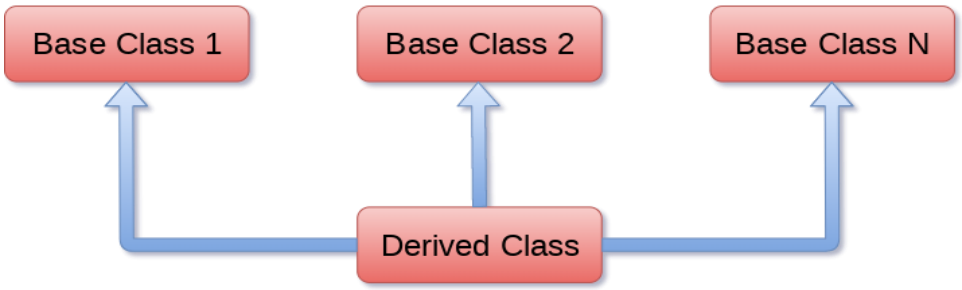
class one:
  def f1(self):
  print("parent class")
class two(one):
  def f2(self):
	parent class
	Intermediate parent class
	child class

```
        print("Intermediate parent class")
class three(two):
    def f3(self):
        print("child class")

ob=three()
ob.f1()
ob.f2()
ob.f3()
```

3. Multiple inheritance

- A child class to inherit from more than one parent class is called multiple inheritance.



Syntax:

```
class parent 1:
        Statement
class parent 2:
        Statement
-----
class parent n:
        Statement

class child(parent1, parent1..... parent n):
        Statement
```

Example :

```
class one:
    def f1(self):
        print("first parent class")
class two:
    def f2(self):
        print("second parent class")
class three(one,two):
    def f3(self):
        print("child class")
```

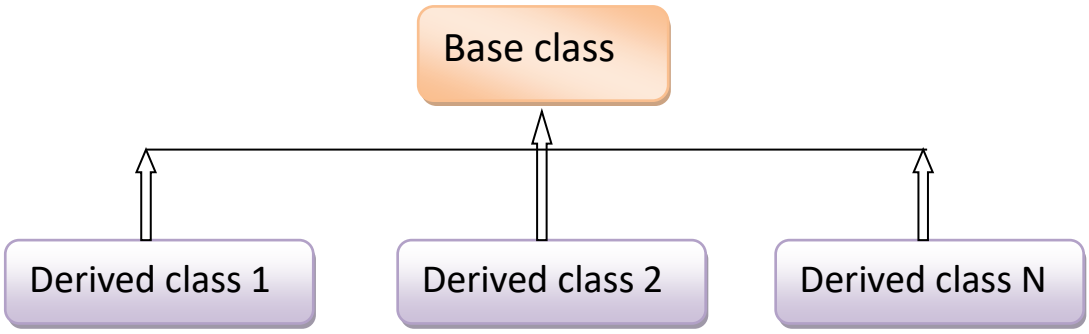
Output:

```
first parent class
second parent class
child class
```

ob=three() ob.f1() ob.f2() ob.f3()	

4. Hierarchical inheritance

- This inheritance allows a class to host as a parent class for more than one child class or subclass.



Syntax:

```
class parent:
        Statement
class child1(parent):
        Statement
class child2(parent):
        Statement
-----
class childN(parent):
        Statement
```

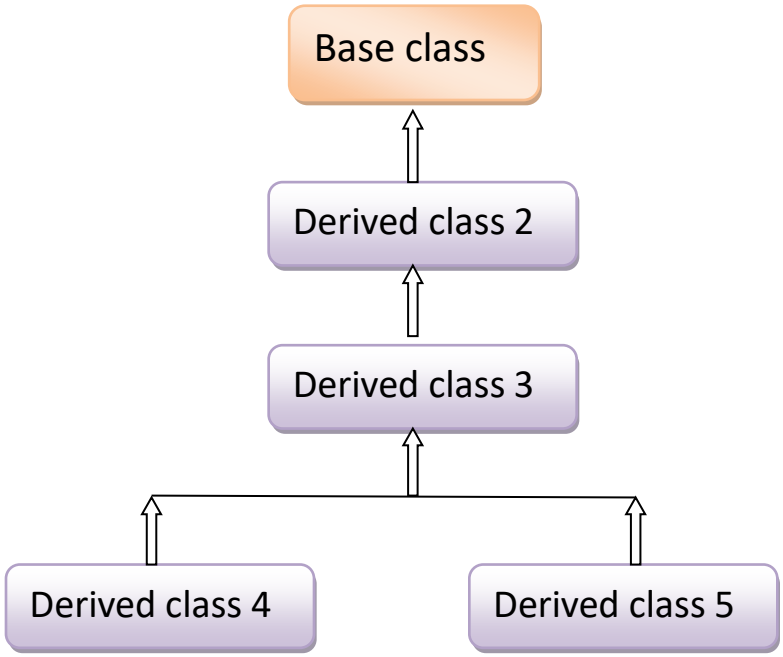
Example :	**Output:**

class one:
  def f1(self):
  print("parent class")
class two(one):
  def f2(self):
  print("first child class")
class three(one):
  def f3(self):
  print("second child class")
	parent class
	first child class
	parent class
	second child class

ob=two() ob.f1() ob.f2() ob1=three() ob1.f1() ob1.f3()	

5. Hybrid inheritance

- Combination of more than one inheritance is called hybrid inheritance.



Example :	**Output:**

class one:
  def f1(self):
  print("first parent class")
class two:
  def f2(self):
  print("second parent class")
class three(two):
  def f3(self):
  print("child class one")
class four(one,three):
  def f4(self):
  print("child class two")
	first parent class
	second parent class
	child class one
	child class two

ob=four() ob.f1() ob.f2() ob.f3() ob.f4()	

4.2.4 Polymorphism

- The word *polymorphism* derives from Greek meaning "something that takes many forms."
- It means that the same function name can be used for different types.
- Types of polymorphism
    - Compile time polymorphism or Static polymorphism
    - Run time polymorphism or dynamic polymorphism
- Method is invoked at compile time is called compile time polymorphism. Ex. Method overloading, Operator overloading
- Method is invoked at runtime is called run time polymorphism. Ex. Method overriding.

Built in polymorphism in python

```
a = 23
b = 11
c = 9.5
s1 = "Hello"
s2 = "There!"
print(a + b)
print(b + c)
print(s1 + s2)
```

Output:

```
34
20.5
HelloThere!
```

```
str = 'HiThere'
tup = ('Mon','Tue','wed','Thu','Fri')
lst = ['Jan','Feb','Mar','Apr']
dict = {'1D':'Line','2D':'Triangle','3D':'Sphere'}
print(len(str))
print(len(tup))
print(len(lst))
print(len(dict))
```

Output:

```
7
5
4
3
```

Method Overriding

- Methods in the child class that have the same name as the methods in the parent class is known as method overriding.

Example :	**Output:**

class one:
  def f1(self):
  print("Good morning")
class two(one):
  def f1(self):
  print("Good afternoon")
class three(one):
  def f1(self):
  print("Good evening")
ob1=one()
ob2=two()
ob3=three()
for a in (ob1,ob2,ob3):
  a.f1()
	Good morning
	Good afternoon
	Good evening

Operator overloading

- Operator overloading in Python is the ability of a single operator to perform more than one operation based on the class (type) of operands.
- For e.g: To use the + operator with custom objects you need to define a method called __add__.

Example :	**Output:**

class one:
  def __init__(self,a,b):
  self.a=a
  self.b=b
  def __add__(self,other):
  a=self.a+other.a
  b=self.b+other.b
  ob3=one(a,b)
  return ob3
 	
ob1=one(90,80)
ob2=one(40,30)
	130
	110

```
ob3=ob1+ob2
print(ob3.a)
print(ob3.b)
```

Method overloading

- **Method Overloading** is a way to create multiple methods with the same name but different arguments. But Python not support method overloading directly. But indirectly support method overloading.

Example :

```
class over:
    def sum(self, a = None, b = None, c = None):
        s = 0
        if a != None and b != None and c != None:
            s = a + b + c
        elif a != None and b != None:
            s = a + b
        else:
            s = a
        return s

ob=over()
print(ob.sum(1))
print(ob.sum(5, 5))
print(ob.sum(10, 2, 3))
```

Output:

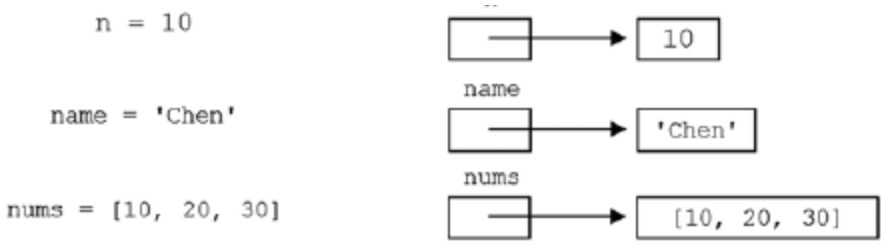
```
1
10
15
```

4.3 Object References

Definition of object : An **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior.

Definition of Object references: A **reference** is a value that references, or "points to," the location of another entity. In Python, objects are represented as a *reference* to an object in memory.

Definition of Garbage collection: Garbage collection is a method of determining which locations in memory are no longer in use, and de allocating them.



Object References to Python Values

- The value that a reference points to is called the **dereferenced value** .

  Ex: a,b,c=10,10,20
  id(a) -> 1682691264
  id(b) -> 1682691264
  id(c) -> 1682691426
- The dereferenced values of a and b, 10, is stored in the same memory location (1682691264), whereas the dereferenced value of c, 20, is stored in a different location (1682691426).
- Even though n and k are each separately assigned literal value 10, they reference the *same instance* of 10 in memory (505498136).
- This saves memory and reduces the number of reference locations that Python must maintain.

4.4 Turtle Graphics

Definition: Turtle graphics refers to a means of controlling a graphical entity (a "turtle") in a graphics window with x,y coordinates.

- Python provides the capability of turtle graphics in the turtle Python standard library module.
- There may be more than one turtle on the screen at once.
- Each turtle is represented by a distinct object. Thus, each can be individually controlled by the methods available for turtle objects.

4.4.1 Creating a Turtle Graphics Window

- import turtle module
- turtle graphics methods called in the form **turtle.** ***methodname*** .
- **setup()** - creates a graphics window of the specified size (in pixels).
- **Screen()** –set the title of the window.
- **bgcolor('color')** - The background color of the window can be changed
- Example : turtle.setup(800,600)

- Window of size 800 pixels width by 600 pixels height is created.
- The center point of the window is at coordinate (0,0).
- x-coordinate values to the right of the center point are positive values, and left are negative values.
- y-coordinate values above the center point are positive values, and below are negative values.

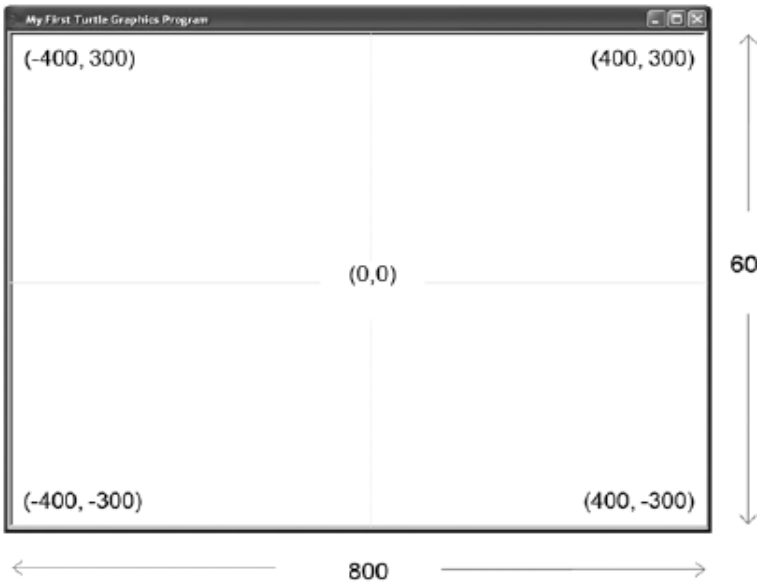
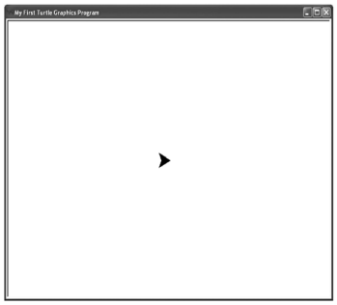


FIGURE 6-18 Python Turtle Graphics Window (of size 800 × 600)

4.4.2 The "Default" Turtle

- A "turtle" is an entity in a turtle graphics window
- **getturtle()** - returns the reference to the default turtle.
- The initial position of all turtles is the center of the screen at coordinate (0,0)
- The default turtle shape is an arrowhead.



4.4.3 Fundamental Turtle Attributes and Behavior

- Turtle objects have three fundamental attributes:
    1. position,
    2. heading (orientation)

3. pen attributes.

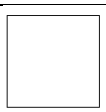
Position

- turtle's position can be changed using *absolute positioning* by use of method setposition().
- hideturtle() - The turtle is made invisible

Example:

```
import turtle

t=turtle.getturtle()
t.hideturtle()
t.setposition(100,0)
t.setposition(100,100)
t.setposition(0,100)
t.setposition(0,0)
```



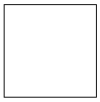
Heading and Relative Positioning

- A turtle's position can also be changed through *relative positioning* .
- A turtle's heading can be changed by turning the turtle a given number of degrees left, left(90), or right, right(90).
- forward() - Moves the turtle forward by the specified amount
- backward()- Moves the turtle backward by the specified amount
- left() - Turns the turtle counter clockwise based on angle
- right() - Turns the turtle clockwise based on angle

Example:

```
import turtle

t=turtle.getturtle()
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
```



Pen Attributes

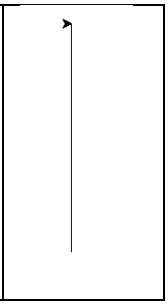
- The pen attribute of a turtle object is related to its drawing capabilities.

- attributes is whether the pen is currently “up” or “down,” controlled by methods penup() and pendown().
- penup()- Picks up the turtle’s Pen
- pendown()-Puts down the turtle’s Pen
- color()-Changes the color of the turtle’s pen
- fillcolor()-fill the shapes with color
- pensize() - determines the width of the lines drawn

Example:

```
import turtle

t=turtle.getturtle()
t.penup()
t.setposition(0,0)
t.pendown()
 t.setposition(0,250)
```



4.4.4 Additional Turtle Attributes

Turtle visibility

hideturtle() – invisible of the turtle

showturtle() - visible of the turtle

Turtle size

turtlesize(width,length)– change the size of the turtle based on width and length.

Turtle Speed

speed(value) - To set the speed of the turtle. Range of speed values from 0 to 10.

The following speed values can be set using a descriptive rather than a numeric value,

10: 'fast' , 6: 'normal' , 3: 'slow' , 1: 'slowest' , 0: 'fastest'

Turtle Shape

shape('value')- shape of the turtle can be changed. value may be 'arrow', 'turtle', 'circle', 'square', 'triangle' and 'classic'.

fillcolor('color') – filled the color in the shape. Color can be red, blue, green, etc.

default shape of turtle is arrow and fill color is black.

Example:	
import turtle t=turtle.getturtle() t.turtlesize(2,5) t.shape('triangle') t.fillcolor('green') t.speed(5)	

4.4.5 Creating Multiple Turtles

- To create and control any number of turtle objects.
- To create a new turtle, the **Turtle()** method is used.
- turtle1 = turtle.Turtle()
- turtle2 = turtle.Turtle()

Example:	**Output:**

import turtle
t = turtle.Turtle()
win=turtle.Screen()
win.setup(500,400)
 	
t.pencolor('blue')
t.pensize(4)
t.shape('turtle')
 	
#draw circle with radius 60 pixels
t.circle(60)
t.clear()
 	
#draw square
for i in range(4):
  t.left(90)
  t.forward(100)
t.clear()
 	
#draw triangle
for i in range(3):
  t.left(120)
  t.forward(100)