Unit 4

OBJECT ORIENTED PROGRAMMING

Akshata S Bhayyar, Asst. Prof. Dept. of CSE, RIT

Basics

- Python is an object-oriented programming language.
- It was developed in 1960 as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.
- Up to now, most of the programs we have been writing use a procedural programming paradigm.
- In procedural programming the focus is on writing functions or procedures which operate on data.
- In object-oriented programming the focus is on the **creation of objects** which contain both data and functionality together.

Basics

- Usually, each **object definition** corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

User-defined compound data types

- Consider the concept of a mathematical point.
- In two dimensions, a point is two numbers (coordinates) that are treated collectively as a **single object.**
- 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.
- Some of the typical operations that one associates with points might be:
  - calculating the distance of a point from the origin, or from another point,
  - finding a midpoint of two points, or
  - asking if a point falls within a given rectangle or circle.

User-defined compound data types

```
class Point:
    """ Point class represents and manipulates x,y coords. """

    def __init__(self):
        """ Create a new point at the origin """
        self.x = 0
        self.y = 0
```

User-defined compound data types

- There is a header which begins with the keyword, class, followed by the name of the class, and ending with a colon.
- Indentation levels tell us where the class ends.
- If the first line after the class header is a string, it becomes the docstring of the class, and will be recognized by various tools.
- Every class should have a method with the special name **__init__.**
- This initialize method is automatically called whenever a new instance of Point is created.
- It gives the programmer the opportunity to set up the attributes required within the new instance by giving them their initial state/values.
- The self parameter (we could choose any other name, but self is the convention) is automatically set to reference the newly created object that needs to be initialized.

User-defined compound data types

```
1  p = Point()          # Instantiate an object of type Point
2  q = Point()          # Make a second point
3
4  print(p.x, p.y, q.x, q.y)  # Each point object has its own x and y
```

This program prints:

```
0 0 0 0
```

A function like Point that creates a new object instance is called a **constructor,** and every class automatically provides a constructor function which is named the same as the class.

User-defined compound data types

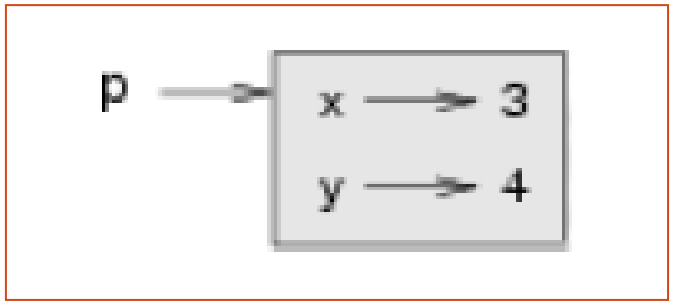
```
class point:
    def __init__(self):
        self.x=0
        self.y=0

p1=point()
p2=point()
print(p1.x,p1.y,p2.x,p2.y)
```

```
0 0 0 0
```

Attributes

- Like real world objects, object instances have both attributes and methods.
- We can modify the attributes in an instance using dot notation:

```
>>> p.x = 3
>>> p.y = 4
```



Attributes

```
class point:
      def __init__(self):
            self.x=0
            self.y=0

p1=point()
p2=point()
print(p1.x,p1.y,p2.x,p2.y)

p1.x=3
p1.y=5
print(p1.x,p1.y,p2.x,p2.y)
```

```
0 0 0 0
3 5 0 0
```

Attributes

We can use dot notation as part of any expression, so the following statements are legal:

```
print("(x={0}, y={1})".format(p.x, p.y))
distance_squared_from_origin = p.x * p.x + p.y * p.y
```

Improving our initializer

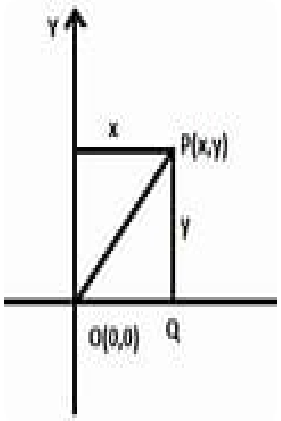
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

Adding other methods to our class

We can add methods to the Point class that are operations for points
Calculate distance from origin

$$AB = \sqrt{x^{2}+y^{2}+z^{2}}$$



Adding other methods to our class

```
class point:
    def __init__(self):
        self.x=0
        self.y=0

    def dist_from_origin(self):
        return((self.x)**2 +(self.y)**2) ** 0.5

p1=point()
p2=point()
print(p1.x,p1.y,p2.x,p2.y)

p1.x=3
p1.y=5
print(p1.x,p1.y,p2.x,p2.y)

dist=p1.dist_from_origin()
print(dist)
```

```
0 0 0 0
3 5 0 0
5.830951
```

Instances as arguments and parameters

- We can pass an object as an argument in the usual way
- Be aware that our variable only holds a reference to an object

Converting an instance to a string

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

  def to_string(self):
    return '({0},{1})'.format(self.x,self.y)



p2=point(5,6)
print(p2.to_string())
```

(5,6)

Instances as return values

Functions and methods can return instances. For example, given two Point objects, find their midpoint.

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

The function creates and returns a new `Point` object:

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = midpoint(p, q)
>>> r
(4.0, 8.0)
```

Instances as return values

Now let us do this as a method instead. Suppose we have a point object, and wish to find the midpoint halfway between it and some other target point:

```
class Point:
    # ...

    def halfway(self, target):
        """ Return the halfway point between myself and the target """
        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2
        return Point(mx, my)
```

```
>>> p = Point(3, 4)
>>> q = Point(5, 12)
>>> r = p.halfway(q)
>>> r
(4.0, 8.0)
```

```
>>> print(Point(3, 4).halfway(Point(5, 12)))
(4.0, 8.0)
```

Difference between Method and Function in Python

Python Method

✓Method is called by its name, but it is **associated to an object** (dependent).

✓A method definition always includes **'self'** as its first parameter.

✓A method is **implicitly passed the object** on which it is invoked.

✓It **may or may not return any data.**

✓A method **can operate on the data (instance variables) that is contained by the corresponding class**

Difference between Method and Function in Python

```
class ABC :
    def method_abc (self):
        print("I am in method_abc of ABC class. ")

class_ref = ABC() # object of ABC class
class_ref.method_abc()
```

Output:

I am in method_abc of ABC class

Difference between Method and Function in Python

Functions

✓Function is block of code that is also **called by its name**. (independent)

✓The function can have different parameters or may not have any at all. If **any data (parameters)** are passed, they are **passed explicitly**.

✓It **may or may not return any data.**

✓Function does not deal with Class and its instance concept.

Difference between Method and Function in Python

```
def Subtract (a, b):
    return (a-b)

print( Subtract(10, 12) ) # prints -2
 print( Subtract(15, 6) ) # prints 9
```

Objects are mutable

We can change the state of an object by making an assignment to one of its attributes.

```
class rect:
      def __init__(s,wi,hi):
            s.w=wi
            s.h=hi

      def add(s,d_wi,d_hi):
            s.w+=d_wi
            s.h+=d_hi

r=rect(10,20)
print(r.w,r.h)
r.add(5,5)
print(r.w,r.h)
```

```
10 20
15 25
```

Sameness

Sameness

To compare the contents of the objects — **deep equality** — we can write a function called same_coordinates:

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

Now if we create two different objects that contain the same data, we can use same_point to find out if they represent points with the same coordinates.

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> same_coordinates(p1, p2)
True
```

Sameness

This type of equality is called shallow equality because it compares only the references, not the contents of the objects.

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 is p2
False
```

```
>>> p3 = p1
>>> p1 is p3
True
```

Sameness

```
p = Point(4, 2)
s = Point(4, 2)
print("== on Points returns", p == s)
# By default, == on Point objects does a shallow equality test

a = [2,3]
b = [2,3]
print("== on lists returns",  a == b)
# But by default, == does a deep equality test on lists
```

This outputs:

```
== on Points returns False
== on lists returns True
```

Copying

✓Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

✓To copy a simple object like a Point, which doesn't contain any embedded objects, copy is sufficient. This is called shallow copying.

```
>>> import copy
>>> p1 = Point(3, 4)
>>> p2 = copy.copy(p1)
>>> p1 is p2
False
>>> same_coordinates(p1, p2)
True
```

Pure Functions

Akshata S Bhayyar, Asst. Prof. Dept. of CSE, RIT

Pure Functions

- A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument or outputting something.
- The only result of calling a pure function is the return value.
- Examples of pure functions are strlen(), pow(), sqrt() etc.
- Examples of impure functions are printf(), rand(), time(), etc.

Generalization

- In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract;
- our intuition for dealing with times is better.
- But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (convertToSeconds and makeTime),
- we get a program that is shorter, easier to read and debug, and more reliable.
- It is also easier to add features later. For example, imagine subtracting two Times

  to find the duration between them.
- The naıve approach would be to implement subtraction with borrowing.
- Using the conversion functions would be easier and more likely to be correct.

INHERITANCE

- Inheritance is the ability to define a new class that is a modified version of an existing class.
- The primary advantage of this feature is that you can add new methods to a class without modifying the existing class.
- It is called inheritance because the new class inherits all of the methods of the existing class.
- Extending this metaphor, the existing class is sometimes called the **parent class.**
- The new class may be called the **child class** or sometimes subclass.
- Inheritance is a powerful feature
- Also, inheritance can facilitate **code reuse**, since you can customize the behaviour of parent classes without having to modify them.
- In some cases, the inheritance structure reflects the natural structure of the problem, which makes the **program easier to understand**.

INHERITANCE

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

x = Person("John", "Doe")
x.printname()
```

John Doe

INHERITANCE

- Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
    pass
```

- Use the Student class to create an object, and then execute the printname method:

INHERITANCE : Create a Child Class

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Person("John", "Doe")
x.printname()

x = Student("Mike", "Olsen")
x.printname()
```

```
John Doe
Mike Olsen
```

INHERITANCE : Add the __init__() Function

- ✓ We want to add the `__init__()` function to the child class
- ✓ When you add the __init__() function, the child class will no longer inherit the parent's __init__() function.
- ✓ The child's __init__() function **overrides** the inheritance of the parent's __init__() function.

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

INHERITANCE : Add the __init__() 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):
    Person.__init__(self, fname, lname)

x = Student("Mike", "Olsen")
x.printname()
```

```
Mike Olsen
```

Use the super() Function

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

Use the 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):
    super().__init__(fname, lname)

x = Student("Mike", "Olsen")
x.printname()
```

```
Mike Olsen
```

Add Properties

```
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):
    super().__init__(fname, lname)
    self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

```
2019
```

Add method

- Add a property called graduationyear to the Student class:

```
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 welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)

x = Student("Mike", "Olsen", 2019)
x.welcome()
```

```
Welcome Mike Olsen to the class of 2019
```

Operator overloading

- Some languages, including Python, make it possible to have different meanings for the same operator when applied to different types.
- **For example,** + in Python means quite different things for integers and for strings. This feature is called operator overloading.
- It is especially useful when programmers can also overload the operators for their own user defined types.

Operator overloading

```
class A:
    def __init__(self, a):
        self.a = a

    # adding two objects
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("Geeks")
ob4 = A("For")

print(ob1 + ob2)
print(ob3 + ob4)

# Actual working when Binary Operator is used.
print(A.__add__(ob1 , ob2))
print(A.__add__(ob3,ob4))
#And can also be Understand as :
print(ob1.__add__(ob2))
print(ob3.__add__(ob4))
```

```
3
GeeksFor
3
GeeksFor
3
GeeksFor
```

Operator overloading

Binary Operators:

Operator	Magic Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Operator overloading

Comparison Operators:

Operator	Magic Method
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Unary Operators:

Operator	Magic Method
-	__neg__(self)
+	__pos__(self)
~	__invert__(self)

Polymorphism

- The word polymorphism means having many forms.
- In programming, polymorphism means the same function name being used for different types.
- The key difference is the data types and number of arguments used in function.

Polymorphism

```
# Python program to demonstrate in-built polymorphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Polymorphism

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output

```
5
9
```

Polymorphism with class methods:

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")
```

```
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance:

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")
```

```
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```