Inheritance in Python

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Summary – Classes and Objects

- We have seen composite types can be created by using objects of component types - i.e. a new class is defined using other class definitions
  - With this, an object of composite class has objects of component classes
  - The relationship between the types is '***has***' , i.e. a class C has objects of class D
- We will now look at another way how class definitions can be used in defining new classes - a very different type of relationship between classes, where a new class *inherits* from another class
- Inheritance is a fundamental property and strength of OO
- It is an advanced concept - we will just familiarize you with it so you understand (can come in useful when using modules)

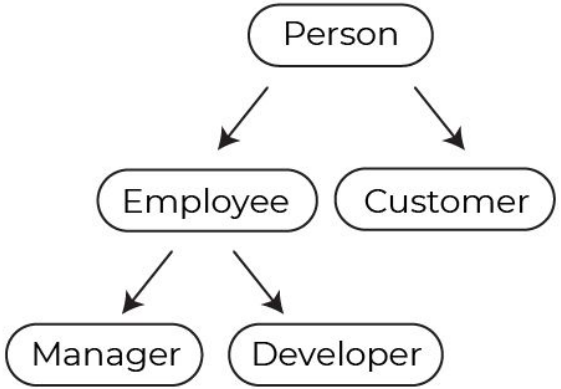
Inheritance

- Motivation - we have a class definition; we want to define a similar class which has more methods and more attributes
- We can define a new class, then the two classes are independent - and we will have to code the new class independently. Changes made in one class on parts that are common to both need to be copied over in the other class
- We can take the existing class, and "refine" it: by borrowing attributes and methods, and adding some more
- This helps in reusing the definitions of the existing class - a very useful property when writing big code
- Inheritance provides this facility

Inheritance

- With inheritance we define a new class using an existing class definition and specifying some modifications/enhancements to it
- I.e. we derive a new type (class) from an existing type (class)
- The new class is called derived class, child class or subclass, and the one from which it inherits is called the base class or parent/super class
- Inheritance also define a class hierarchy - this hierarchy allows modeling of many real life hierarchies easily in programs
- We will mostly discuss two-level hierarchy only

Inheritance as an image



- Employee and Customer are child classes of Person
- Manager and Developer are child classes of Employee
- We can also say Manager and Developer classes are inherited from Person

Inheritance Syntax

```
class BaseClass:
    <body of base class>
    # attributes and methods

class DerivedClass (BaseClass):
    <body of derived class>
```

- The DerivedClass specifies the BaseClass it is inheriting from
- On inheriting, all attributes and methods of BaseClass are available to this class
- DerivedClass can add new features and new methods
- DerivedClass is said to *extend* the BaseClass
- (A class can inherit from multiple classes - we will not discuss it)

Objects of Base/Derived class

- A derived class can define additional attributes and methods
- An object of derived class has all attributes of base class, and can perform all operations of base class; + it has attributes of derived class and all its methods (*extends* the base class)
- So an object of derived class is also on object of base class - as has all the base class attributes and can perform those methods
- But an object of base class is not an object of derived class
- This defines an **"is-a"** relationship between the classes - an object of derived class is an object of the base class also, but not the other way around

Objects of Base/Derived class

- Creating an object of any base class is as with regular class
- Creating an object of derived class - the attributes of the base class automatically get defined, and methods become available
  - Often __init__() of base class called to initialize that part of state
- When an attribute is accessed/method performed on the object, python first searches in the derived class - if it exists that is used
- If it does not exist, then it searches in the base class - if it exists then it is used (this is applied recursively)
- (So, attributes/methods defined in derived class get selected over those defined in base class, if they have same names)

An Example

- Consider a polygon - it is a general type of geometric structure
- Then there are specialized polygons - triangle, quadrilateral, hexagon, octagon, …
- There are some common properties of a polygon - e.g. the number of sides it has, length of each side, perimeter
- Then there are some special features for different types of polygons - e.g. area for each polygon is a different formula, and other types of polygons may need more information (e.g. angles)
- We can define polygon as a base class, and defined inherited classes for specialized polygons
- Let’s start with defining a polygon and inherit it to define a triangle

Polygon – Triangle

```
class Polygon:
    def __init__(self, num):
        self.nosides = num
        self.sides = [0]*num

    def setsides(self, l):
        if self.nosides != len(l):
            print ("Error-no of sides not correct")
            return
        for i in range(self.nosides):
            self.sides[i] = l[i]

    def perim(self):
        return sum(self.sides)
```

- Polygon is the base class - has *nosides* and *sides* as the main attributes
- Polygon is created by specifying the nosides, and initializing a list of sides
- Provides method to set the sides
- A method to compute the perimeter of the polygon
- But nothing to compute area - as there is no general formula

Polygon – Triangle

```
class Triangle(Polygon):
    def __init__(self):
      self.nosides = 3
      self.sides = [0]*3
      # Polygon.__init__(self, 3)

    def area(self):
      s=((self.sides[0]+self.sides[1]+\
                self.sides[2])/2)
      tmp = s*(s-self.sides[0])*\
                (s-self.sides[1])*\
                (s-self.sides[2])
      return math.sqrt(tmp)
```

- Class Triangle inherits from Polygon
- At creation, attributes available - init() sets them
- More natural - call init() of Polygon(Highlighted)
- (You can call a method by class_name.method - the object has to be explicitly passed)
- Defines a new method area

Polygon-Triangle use

```
# Using Triangle


t = Triangle()
t.setsides([4, 5, 6])
print("Perim: ", t.perim())
print("Area: ", t.area())
```

```
Output:

Perim:  15
Area:  9.921567416492215
```

- Object t is of Triangle type
- Note that on t method setsides() from is permitted
- Method perim() from Polygon used
- Method area() of Triangle will be used
- So, all attributes of Polygon and Triangle are available
- dir (t) gives attributes: 'area', 'dispsides', 'nosides', 'perim', 'setsides', 'sides'

Inheritance: Base Class, Derived Class

- A class definition can inherit (and extend) another class's definition
  - Base class: Whose definition is being inherited (extended)
  - Derived class: The class which is derived using the base class
- On inheriting, all attributes and all methods of base class are automatically defined for the objects of the derived class
- A derived class can add new attributes, as well as new methods
- Objects can be created of base class or derived class
  - Methods of base class - avail on objects of derived class
  - New methods of derived class not there for base class obj
- So objects of derived class are also objects of base class, but not vice versa

Quiz – Single correct

Which of the following is the output of the code?

A. 9, 2
B. 10,2
C. 11,2
D. 12,2

```
class A:
    def __init__(self, x):
        self.x = x
    def incr(self, x):
        self.x += x

class B(A):
    def __init__(self, x, y):
        A.__init__(self, x)
        self.y = y
    def decr(self, x):
        self.x -= 2

o = B(10,2)
o.incr(1)
o.decr(o.y)
print(o.x, o.y)
```

Quiz – Single correct

Which of the following is the output of the code?

A. **9, 2**

B. 10,2

C. 11,2

D. 12,2

```
class A:
    def __init__(self, x):
        self.x = x
    def incr(self, x):
        self.x += x

class B(A):
    def __init__(self, x, y):
        A.__init__(self, x)
        self.y = y
    def decr(self, x):
        self.x -= 2

o = B(10,5)
o.incr(1)
o.decr(o.y)
print(o.x, o.y)
```

Recap

- Looking at defining a new class (type) by inheriting from an existing class (type) definition
- New class inherits all attributes and methods, and can add more

Quiz – Numerical

What is the output of the following code?

```
class A:
    def __init__(self, x):
        self.x = x
    def incr(self, x):
        self.x += x

class B(A):
    def __init__(self, x, y):
        A.__init__(self, x)
        self.y = y
    def decr(self, x):
        self.x -= x
        self.y -= x

o = B(10,5)
o.incr(o.x)
o.decr(o.y)
print(o.x + o.y)
```

Quiz – Numerical

What is the output of the following code?

15

```
class A:
    def __init__(self, x):
        self.x = x
    def incr(self, x):
        self.x += x

class B(A):
    def __init__(self, x, y):
        A.__init__(self, x)
        self.y = y
    def decr(self, x):
        self.x -= x
        self.y -= x

o = B(10,5)
o.incr(o.x)
o.decr(o.y)
print(o.x + o.y)
```

Derived class can redefine

- For a attribute/method - the given ‘derived’ class is first searched, base class searched only if not found in class
- I.e. if a method is defined both in subclass and parent class - for objects of subclass, the method for the **subclass** will be executed
- Redefinition of methods can lead to subtle issues - it has to be done very carefully and generally to be avoided
- Exception is the init() method - it will be there in both; often init() method of subclass will invoke init method of parent
- We will assume that subclass specializes by adding new attributes and methods, and do not redefine anything (except __init__)

Derived class should be a "specialization"

- New methods/attributes can be defined, and base class methods can be redefined
- Derived class should be such that objects of this class can be used anywhere objects of base class can be used
- I.e. methods of base class should not be redefined in a manner that the behavior changes (then when replacing a base class object with derived class obj - the method will behave differently)
- Called Liskov's substitution principle: objects of superclass can be replaced by objects of a subclass without breaking the application
  - With this, a derived class object is a more "specialized" object
  - E.g. a triangle is a specialized polygon

Use of super()

- In a subclass code we can use super() to refer to the parent class, I.e. in subclass declaration, we need to explicitly mention the base class, but in the body can use super() to refer to it
- super() provides a proxy object, through which all methods of the base class can be called
- With super() don't have to remember base class name exactly, if base class name is changed - have to change only in one place…
- You will often see the use of super() in python code, E.g.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self, 3)   # approach 1 - needs to know name
        super().__init__(3)   # approach 2 - no need to remember name
```

Polygon

```
class Polygon:
    def __init__(self, num):
        self.nosides = num
        self.sides = [0]*num

    def setsides(self, l):
        if self.nosides != len(l):
            print ("Error-sides not correct")
            return
        for i in range(self.nosides):
            self.sides[i] = l[i]

    def perim(self):
        return sum(self.sides)
```

- Polygon is the base class - has *nosides* and *sides* as the main attributes
-

Example – Polygon – Quadrilateral

```
class Quad(Polygon):
    def __init__(self):
     super().__init__(4)
    def setdiag(self, diag):  #Diagonal between 1st 2 sides
     self.diag = diag
    def area(self):
     s = (self.sides[0]+self.sides[1]+self.diag)/2
     tmp = s*(s-self.sides[0])*(s-self.sides[1])*(s-self.diag)
     a1 = math.sqrt(tmp)
     s = (self.sides[2]+self.sides[3]+self.diag)/2
     tmp = s*(s-self.sides[2])*(s-self.sides[3])*(s-self.diag)
     a2 = math.sqrt(tmp)
     return a1+a2
```

- For quad, you need one more attribute - one diagonal (to compute the area)
- Dir: 'area', 'dispsides', 'nosides', 'perim', 'setdiag', 'setsides', 'sides', 'diag'

Quad…

```
q = Quad()
q.setsides([3, 4, 7, 8])
q.setdiag(5)
print("Perim: ", q.perim())
print("Area: ", q.area())
```

```
Perim:  22
Area:  23.320508075688775
```

Quiz – Single correct

Which of the following is the output of the code?

A. Runtime Error
B. Syntax is wrong
C. hey dude
D. yo dude

```
class ques:
    def __init__(self, x="hey"):
        self.x = x

class another(ques):
    def __init__(self, y="dude"):
        super().__init__("yo")
        self.y = y

obj = another()
print(obj.x, obj.y)
```

Quiz – Single correct

Which of the following is the output of the code?

A. Runtime Error
B. Syntax is wrong
C. hey dude
D. yo dude

```
class ques:
    def __init__(self, x="hey"):
        self.x = x

class another(ques):
    def __init__(self, y="dude"):
        super().__init__("yo")
        self.y = y

obj = another()
print(obj.x, obj.y)
```

Multiple Child Classes

- A base class can be inherited by many other class definitions
- I.e. there can be many subclasses of a class
- All should follow Liskov's substitution principle
- Have already seen an example - Polygon as base class, two subclasses - Triangle and Quad
- Subclasses have no relationship among themselves - e.g. object of a subclass cannot be used in place of obj of another subclass
- Subclasses are related only with the parent

Another Example

- A common use - define a general person, then for different applications create specialized subclasses
- Let us look at person with some basic attributes like name, address, maybe DOB, …
- A specialized subclass student, and another Patient - both have some other attributes and some other methods
  - I.e. a student or patient is a specialized person with more attributes, and more methods
- Person class will have methods for the attributes it has, Student and Patient need to support methods that are special to them

Example...

```
class Person:
    def __init__(self, name, addr):
        self.name = name
        self.addr = addr
    def getname(self):
        return self.name
    def changeinfo(self, name, addr):
        self.__init__(name, addr)
        __init__(self, name, addr)
    def __str__(self):
        return f'Person: {self.name};
{self.addr}'
```

```
class Student(Person):
    def __init__(self, name, addr, rollno):
        super().__init__(name, addr)
        self.rollno = rollno
    def setcgpa(self, num):
        self.cgpa = num
    def gradyr(self, yr):
        self.gradyr = yr
    def __str__(self):
        s = super().getname()
        s = "Student: " + s+ " " +
str(self.rollno) +" " + str(self.cgpa)
        return s
```

Example...

```
class Patient(Person):
    def __init__(self, name, addr, age):
        super().__init__(name, addr)
        self.age = age
    def diagnosis(self, disease):
        self.disease = disease
    def prescription(self, s):
        self.rept = s
    def __str__(self):
        return f'Patient: {self.name}, {self.age}, {self.disease}, {self.rept}'

s1 = Student("A", "xxx", 11)
s1.setcgpa(8.5)
print(s1)
p1 = Patient("B", "yyy", 55)
p1.diagnosis("covid")
p1.prescription("Rest and isolation")
print(p1)
```

```
Student: A 11 8.5
Patient: B, 55, covid, Rest and isolation
```

Example...

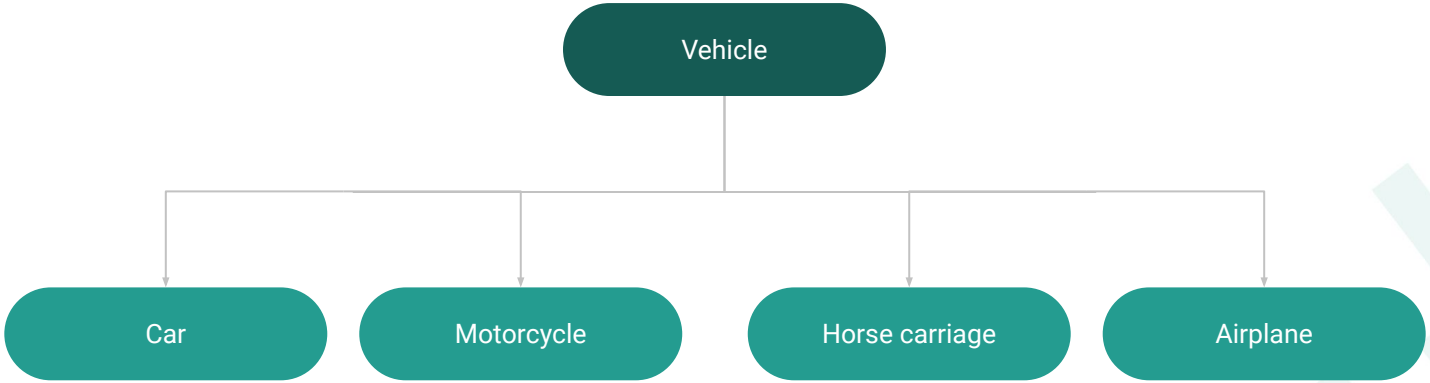
```
p0 = Person("P", "PPP")

l = [p0, s1, p1]
for i in l:
    print("The person is a : ", type(i), "Info: ")
    print(i)
```

```
The person is a :  <class '__main__.Person'> Info:
Person: P; PPP

The person is a :  <class '__main__.Student'> Info:
Student: A 11 8.5

The person is a :  <class '__main__.Patient'> Info:
Patient: B, 55, covid, Rest and isolation
```

Example – Vehicle

There can be a base vehicle class. This can have methods like start and move along with attributes like fuel and fuel capacity.

From this base class multiple subclasses of Car, Motorcycle, Horse Carriage, Airplane can be derived.



Converting objects of base type to derived

- An object of derived type is automatically an object of base type - so operations of base type can be applied
- Derived type objects are "specialized" version of base type objs
- How to convert objects of base type to objects of derived type - e.g. how to convert a person object to a student (or patient) obj
- We will have to add the extra attributes if this is done
- There is no clean way of doing this; the most direct way is to change the __class__ attribute, e.g.

```
p0.__class__ = Student
```

- It is best to avoid this type of conversion - better to create a new object and copy from base object

Quiz – Single Correct

Which of the following is the output of the code?

A. In class A
B. In class B
C. In class C
D. In class D

```
class A:
    def f(self):
        print(" In class A")
class B(A):
    def f(self):
        print(" In class B")
class C(A):
    def f(self):
        print("In class C")
class D(B, C):
    def f(self):
        print("In class D")

r = D()
r.f()
```

Quiz – Single Correct

Which of the following is the output of the code?

A. In class A
B. In class B
C. In class C
D. In class D

```
class A:
    def f(self):
        print(" In class A")
class B(A):
    def f(self):
        print(" In class B")
class C(A):
    def f(self):
        print("In class C")
class D(B, C):
    def f(self):
        print("In class D")

r = D()
r.f()
```

Abstract Class / Interfaces

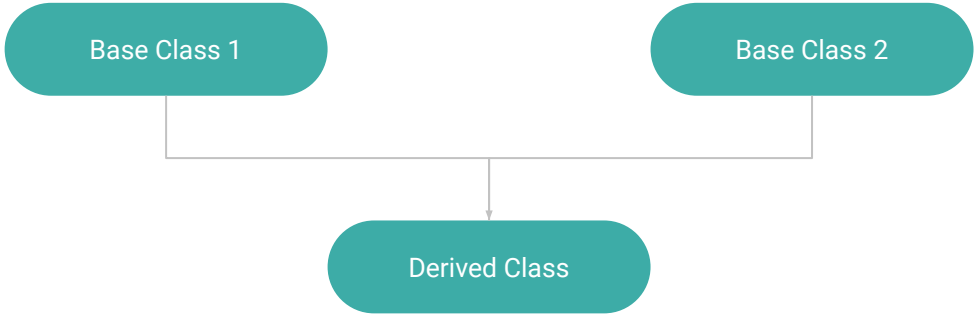
- An advanced concept and practice - only familiarizing you
- A base class can define some abstract methods: they defines the interface, but there is no body to execute it
- An abstract class is one which has some such methods
- Abstract classes **require** subclasses to redefine the abstract method with actual implementation
- I.e. an abstract class defines an interface (methods with their interfaces), which ensures that all child classes have these methods with the defined interface
- Often used for the purpose of defining standard API / interfaces

Example for Abstract Class/interfaces

- Let’s revisit the multiple inheritance example from earlier.
- The polygon class can define an abstract method - area().
- Then all subclasses will be required to implement this method when they inherit from the Polygon class
- I.e. all of the subclasses can then have their own definition of the area() - as appropriate for that type of polygon - but all must have
- (Using it requires abc (abstract base class) to be included - then some functions have to be inherited)

Multiple Inheritance

- A class definition can inherit from multiple other classes, i.e. a class can be derived from multiple base classes
- All the attributes and methods of all base classes will be inherited
- Subtle issues come up when multiple inheritance is there
- Advanced concept - we will not go into it



Feedback – Color Scale

How attentive/engaged you were in today's lecture?

- Red - Attention/engagement was lesser
- Yellow - Was about the same
- Green - Attention/engagement was more

Feedback – Color Scale

How well do you feel the concepts taught today were understood by you?

Answer in three options :

- Red - Low Understanding
- Yellow - Medium understanding
- Green - Good understanding

Summary

- Inheritance is another approach of defining a new class (type) using another class's definition
- On inheriting from a base class, a derived class inherits all the attributes and methods of the base class
- The derived class can add new attributes and methods - thereby "specializing" the base class
- Hence objects of derived class are also objects of base class (but not the other way around)
- Inheritance allows reusing class definitions and allows modeling of hierarchies that naturally exist in real world

Next class - online