Computational Thinking with Programming

python™

Lecture - 23

Inheritance and Polymorphism

BENNETT UNIVERSITY
TIMES OF INDIA GROUP

@cse_bennett @csebennett

Today…

- Last Session:
  - Encapsulation.

- Today's Session:
  - Inheritance.
  - Polymorphism

- Hands on Session with Jupyter Notebook:
  - We will practice on the objects programming in Jupyter Notebook.

Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

OOP Inheritance

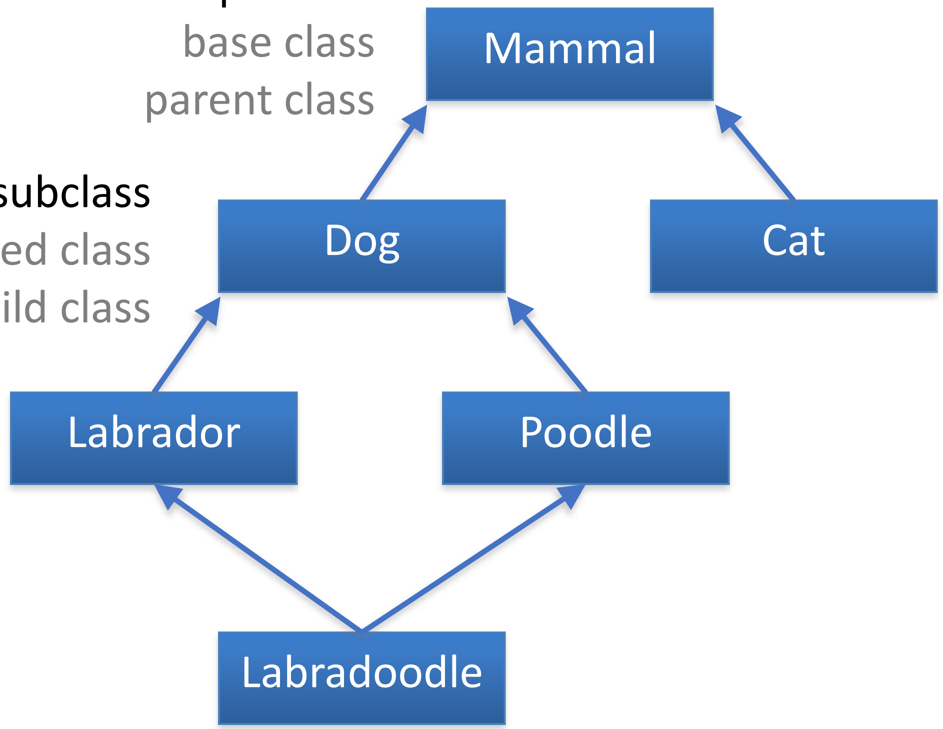
- "Inheritance is the ability to define a new class that is a modified version of an existing class." – Allen Downey, *Think Python*
- "A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of superclasses." – Grady Booch, *Object-Oriented Design*

OOP Inheritance (cont.)

- Conceptual example:

superclass
base class
parent class

subclass
derived class
child class



single inheritance

class hierarchy

multiple inheritance

Inheritance Syntax

- The syntax for inheritance was already introduced during class declaration
  - **C1** is the name of the subclass
  - **object** is the name of the superclass
  - for multiple inheritance, superclasses are declared as a comma-separated list of class names

```
class C1(object):
    "C1 doc"
    def f1(self):
        # do something with self
    def f2(self):
        # do something with self

# create a C1 instance
myc1 = C1()

# call f2 method
myc1.f2()
```

Inheritance Syntax (cont.)

- Superclasses may be either Python- or user-defined classes
  - For example, suppose we want to use the Python list class to implement a stack (last-in, first-out) data structure
  - Python list class has a method, **pop**, for removing and returning the last element of the list
  - We need to add a **push** method to put a new element at the end of the list so that it gets popped off first

```
class Stack(list):
    “LIFO data structure"
    def push(self, element):
        self.append(element)
    # Might also have used:
    #push = list.append

st = Stack()
print "Push 12, then 1"
st.push(12)
st.push(1)
print "Stack content", st
print "Popping last element", st.pop()
print "Stack content now", st
```

Inheritance Syntax (cont.)

- A subclass inherits all the methods of its superclass
- A subclass can **override** (replace or augment) methods of the superclass
  - Just define a method of the same name
  - Although not enforced by Python, keeping the same arguments (as well as pre- and post-conditions) for the method is highly recommended
  - When augmenting a method, call the superclass method to get its functionality
- A subclass can serve as the superclass for other classes

Overriding a Method

- __init__ is frequently overridden because many subclasses need to both (a) let their superclass initialize their data, and (b) initialize their own data, usually in that order

```
class Stack(list):
    push = list.append

class Calculator(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.accumulator = 0
    def __str__(self):
        return str(self.accumulator)
    def push(self, value):
        Stack.push(self, value)
        self.accumulator = value

c = Calculator()
c.push(10)
print c
```

Multiple Inheritance

- Python supports multiple inheritance
- In the **class** statement, replace the single superclass name with a comma-separated list of superclass names
- When looking up an attribute, Python will look for it in "method resolution order" (MRO) which is approximately left-to-right, depth-first
- There are (sometimes) subtleties that make multiple inheritance tricky to use, eg superclasses that derive from a common super-superclass
- Most of the time, single inheritance is good enough

Class Diagrams

- Class diagrams are visual representations of the relationships among classes
  - They are similar in spirit to entity-relationship diagrams, unified modeling language, *etc* in that they help implementers in understanding and documenting application/library architecture
  - They are more useful when there are more classes and attributes
  - They are also very useful (along with documentation) when the code is unfamiliar

Inheritance Example

```
class student:                # base class
    def __init__(self,name,age):
        self.name = name
        self.age =age

    def get_data(self):
        self.name = input("Enter the name")
        self.age = input("Enter your age")

    def put_data(self):
        print(self.name)
        print(self.age)
```

```
student1 = student("anurag",25)
student1.put_data()
```

```
anurag
25
```

```
class sciencestudent(student):          # child class
    def science(self):
        print("This is science student")
```

```
sciencestudent2 = sciencestudent("mohit",30)
sciencestudent2.put_data()
```

```
mohit
30
```

```
sciencestudent1 = sciencestudent("","")
sciencestudent1.get_data()
sciencestudent1.put_data()
```

```
Enter the name : xyz
Enter your age : 29
    xyz
    29
```

```
sciencestudent2.science()
```

```
This is science student
```

```
student1.science()
```

Example

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

Mike Olsen

Use the pass keyword when you do not want to add any other properties or methods to the class.

Example of Multiple Inheritance

```
class A:
    def A_method(self):
        print("method for A")

class B:
    def B_method(self):
        print("method for B")

class C(A,B):
    def C_method(self):
        print("method for c")

cc = C()
cc.A_method()
cc.B_method()
cc.C_method()
```

```
method for A
method for B
method for c
```

```
class A:
    def A_method(self):
        print("method for A")

class B:
    def A_method(self):
        print("method for B")

class C(A,B):
    def C_method(self):
        print("method for c")

cc = C()
cc.A_method()
cc.A_method()
cc.C_method()
```

```
method for A
method for A
method for c
```

```
class A:
    def A_method(self):
        print("method for A")

class B:
    def A_method(self):
        print("method for B")

class C(A,B):
    def A_method(self):
        print("method for c")

cc = C()
cc.A_method()
cc.A_method()
cc.A_method()
```

```
method for c
method for c
method for c
```

Example of Multiple Inheritance

```
class A:
   def A_method(self):
      print("method for A")

class B:
   def A_method(self):
      print("method for B")

class C(B,A):
   def C_method(self):
      print("method for c")

cc = C()
cc.A_method()          method for B
cc.A_method()          method for B
cc.C_method()          method for c
```

Example of Multilevel Inheritance

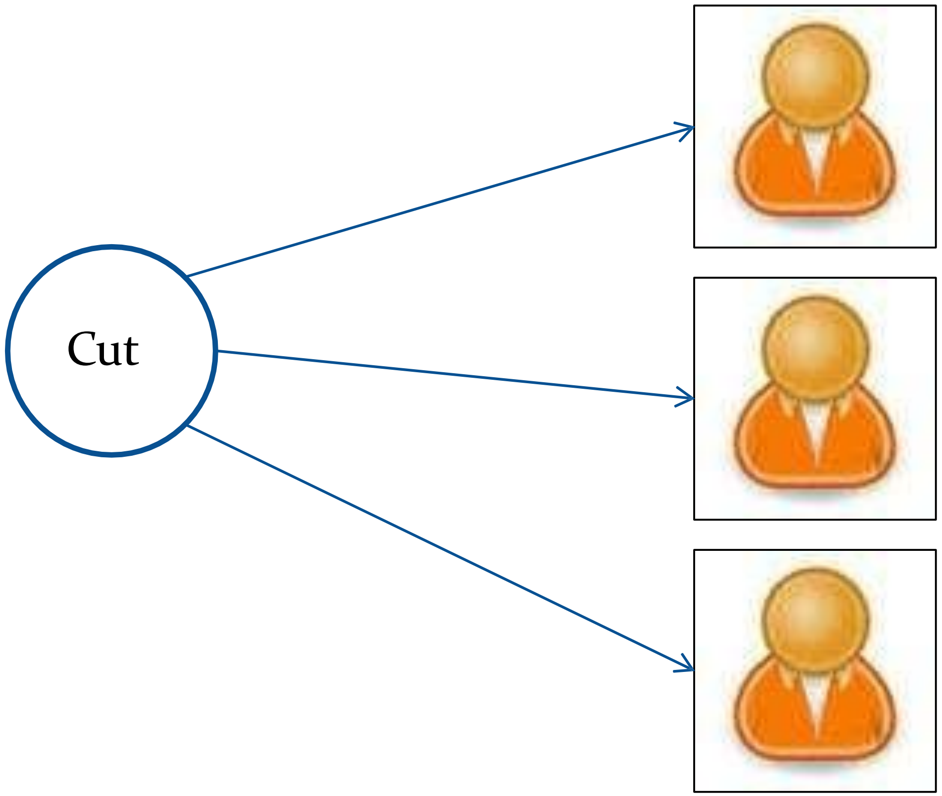
```
class A:
    def A_method(self):
        print("method for A")

class B(A):
    def B_method(self):
        print("method for B")

bb = B()
bb.B_method()                method for B

class C(B):
    def C_method(self):
        print("method for c")

cc = C()
cc.A_method()                method for A
cc.B_method()                method for B
cc.C_method()                method for c
```

Polymorphism

- Polymorphism in Latin word which made up of ***'poly'*** means many and ***'morphs'*** means forms
- From the Greek , Polymorphism means ***many(poly)*** ***shapes*** ***(morph)***
- This is something similar to a word having several different meanings depending on the context
- Generally speaking, polymorphism means that a method or function is able to cope with different types of input.

A simple word ‘Cut’ can have different meaning depending where it is used

If any body says “Cut” to these people



- Surgeon: The Surgeon would begin to make an incision
- Hair Stylist: The Hair Stylist would begin to cut someone’s hair
- Actor: The actor would abruptly stop acting out the current scene, awaiting directional guidance

There are two kinds of Polymorphism

Overloading :

Two or more methods with different signatures

Overriding:

Replacing an inherited method with another having the same signature

Overriding

```
class Parent:       # define parent class
   def myMethod(self):
      print('Calling parent method')

class Child(Parent): # define child class
   def myMethod(self):
      print('Calling child method')


c = Child()          # instance of child
c.myMethod()         # child calls overridden method

p = Parent()
p.myMethod()
```

Calling child method

Calling parent method

Operator Overloading

- Python operators work for built-in classes.

- But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading

- One final thing to mention about operator overloading is that you can make your custom methods do whatever you want.

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

  def __sub__(self,other):
    x = self.x - other.x
    y = self.y - other.y
    return Point(x,y)


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

  def __mul__(self,other):
    x = self.x * other.x
    y = self.y * other.y
    return Point(x,y)

  def __pow__(self,other):
    x = self.x**other.x
    y = self.y**other.y
    return Point(x,y)

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

p1=Point(5,4)
p2=Point(3,2)

print(p1+p2)
print(p1-p2)
print(p1*p2)
print(p1**p2)
```

```
Point (8, 6)
Point (2, 2)
Point (15, 8)
Point (125, 16)
```

Explanation for Operator Overloading Sample Program

What actually happens is that, when you do p1 - p2, Python will call p1._sub_(p2) which in turn is Point._sub_(p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 – p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)

Access Modifiers

- 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.

- Public variables, you do not have to add any prefixes at all.

- ❑ Access modifiers play an important role to protect the data from unauthorized access as well as protecting it from getting manipulated.
- ❑ When inheritance is implemented there is a huge risk for the data to get manipulated due to transfer of unwanted data from the parent class to the child class.
- ❑ Therefore, it is very important to provide the right access modifiers for different data members and member functions depending upon the requirements.

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
```

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self._salary=sal
```

Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed.

protected attribute

In fact, this doesn't prevent instance variables from accessing or modifying the instance.

Hence, the **responsible** programmer would **refrain** from accessing and modifying instance variables prefixed with _ from outside its class.

```
class employee:
  def __init__(self, name, sal):
    self.__name=name  # private attribute
    self.__salary=sal # private attribute
```

Similarly, a double underscore __ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an Attribute Error.

```
e1=Employee("Bill",10000)
Print(e1._Employee__salary)
10000
```

Python performs name mangling of private variables. Every member with double underscore will be changed to object._class__variable. If so required, it can still be accessed from outside the class

Data Model

- A data model is a logic organization of the real world objects (entities), constraints on them, and the relationships among objects.
- A core object-oriented data model consists of the following basic object-oriented concepts:

  (1) object and object identifier: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).

  (2) attributes and methods: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.

  [An attribute is an instance variable, whose domain may be any class: user-defined or primitive. A class composition hierarchy (aggregation relationship) is orthogonal to the concept of a class hierarchy. The link in a class composition hierarchy may form cycles.]

  (3) class: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.

  (4) Class hierarchy and inheritance: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods.

```python
class parent:
    def __init__(self):
        print("parent __init__")

class child:
    def __init__(self):
        print("child __init__")

c = child()
```

```
child __init__
```

```python
class Parent:
    def __init__(self):
        print('Parent __init__')

class Child(Parent):
    def __init__(self):
        print('Child __init__')
        super().__init__()

child = Child()
```

```
child __init__
Parent__init__
```

```python
class Parent:
    def __init__(self, name):
        print('Parent __init__', name)

class Child(Parent):
    def __init__(self):
        print('Child __init__')
        super().__init__('max')

child = Child()
```

```
child __init__
Parent__init__ max
```

Abstract Class

- Abstract classes are classes that contain one or more abstract methods.
- An abstract method is a method that is declared but contains no implementation.
- Abstract classes may not be instantiated and require subclasses to provide implementations for the abstract methods.

```
class AbstractClass:
    def do_something(self):
        pass

class B(AbstractClass):
    pass

a = AbstractClass()
b = B()
```

Our example implemented a case of simple inheritance which has nothing to do with an abstract class.

- ❑ We see that this is not an abstract class, because:

- ❑ we can instantiate an instance from class AbstractClass
- ❑ we are not required to implement do_something in the class definition of B

Abstract Class

- Python on its own doesn't provide abstract classes. Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs).
- This module is called for obvious reasons abc.

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

```
class DoAdd(AbstractClassExample):
    def do_something(self):
        return self.value + 42

class DoMul(AbstractClassExample):
    def do_something(self):
        return self.value * 42

x = DoAdd(10)
y = DoMul(10)

print(x.do_something())          52
print(y.do_something())          420
```

```
class DoAdd(AbstractClassExample):
    pass

x = DoAdd(4)
```

```
TypeError: Can't instantiate abstract class DoAdd with abstract
methods do_something
```

Abstract Class

A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

We may think that abstract methods can't be implemented in the abstract base class. This impression is wrong: An abstract method can have an implementation in the abstract class

Even if they are implemented, designers of subclasses will be forced to override the implementation.

Like in other cases of "normal" inheritance, the abstract method can be invoked with super() call mechanism. This makes it possible to provide some basic functionality in the abstract method, which can be enriched by the subclass implementation.

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
    def do_something(self):
        super().do_something()
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()
```

```
Some implementation!
The enrichment from AnotherSubclass
```

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
    def __init__(self, value):
        self.value = value
        super().__init__()

    def display(self):
        print(self.value)

x= AbstractClassExample(10)
x.display()
```

Persistent storage of objects

- Persistent Storage is any data storage device that retains data after power to that device is shut off.
- It is also sometimes referred to as non-volatile storage.
- Hard disk drives and solid-state drives are common types of persistent storage. This can be in the form of file, block or object storage.

In python, we have a module to store a object into memory. Later, we can used it.

To support storing Python data in a persistent form on disk. The pickle and marshal modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes.

“**Pickling**” is the process whereby a **Python** object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.

```
>>> a1 = 'apple'
>>> b1 = {1: 'One', 2: 'Two', 3: 'Three'}
>>> c1 = ['fee', 'fie', 'foe', 'fum']
>>> f1 = file('temp.pkl', 'wb')
>>> pickle.dump(a1, f1, True)
>>> pickle.dump(b1, f1, True)
>>> pickle.dump(c1, f1, True)
>>> f1.close()

>>> f2 = file('temp.pkl', 'rb')
>>> a2 = pickle.load(f2)
>>> a2
'apple'
>>> b2 = pickle.load(f2)
>>> b2
{1: 'One', 2: 'Two', 3: 'Three'}
>>> c2 = pickle.load(f2)
>>> c2
['fee', 'fie', 'foe', 'fum']
>>> f2.close()
```

Persistent storage of objects

```
import pickle
class Company:
    pass


company1 = Company()
company1.name = 'banana'
company1.value = 40
with open('company.pkl', 'wb') as f:
    pickle.dump(company1, f, pickle.HIGHEST_PROTOCOL)
```

Thank You

?