Introduction to Python

Encapsulation, Abstraction and Inheritance

Topics

1) Encapsulation
2) Interface
3) Abstraction
4) Application Programming Interface(API)
5) Mutator/Accessor Methods
6) Inheritance

Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).

It describes the idea of wrapping data and the methods that work on data within one unit.

Creating custom classes as discussed in the previous lecture is an example of encapsulation.

Interface

Encapsulation is used to hide the values or data of a class, preventing unauthorized parties' direct access to them.

An important idea behind encapsulation is that data inside the object should only be accessed through a public ***interface*** – that is, the object's methods.

What's the interface to your TV?

Remote control/buttons on TV.(buttons correspond to methods())

Your car?

Steering wheel, brake, gas pedal, transmission, etc..(all these correspond to methods())

Interface

More generally, an interface is the public, exposed functionality of a system or program. The data and specific implementation details of the system/program should be hidden.

You don't need to know the details of how your TV or car work to be able to use them.

Abstraction: distance between ideas and implementation. This refers to arranging programming code so that functionality may be separated from specific implementation details.

We can use lists, strings and dictionaries in Python without knowing how those classes are implemented.

More on Abstraction

An **abstraction** has two roles in a program:

1) It reduces or removes details to help you to understand something new. I might not know the details of how append is implemented but I can still understand what it does to a list.

```
In [1]: a = [0, 5, 2]
In [2]: a.append(3)
In [3]: a
Out [3]: [0, 5, 2, 3]
```

2) It helps you manage the details, code and complexity of your program. The following code is easy to write/understand even if move() is a complicated method.

```
for x in enemy_list:
    x.move()
```

Abstractions

There are many layers to any abstraction.

In general, when it comes to abstraction, “less is more”. The less detail, the more abstract.

For example, “An engine helps a car move.” is a statement with very little detail and hence very abstract.

But the statement “In an engine, fuel, air, pressure, and electricity come together to create the small explosion that moves the car's pistons up and down, thus creating the power to move the vehicle.” has more details and is less abstract.

Abstractions

There are many layers to any abstraction: higher-level abstraction is more general; lower-level is more specific.

Python, Javascript, Java are a high-level programming languages. You can write games easily without know how hardware/memory works.

Assembly is a low-level programming language. You have to know the details of how the operating system/memory works. Writing a game can take many lines of code.

Lower-level abstractions can be combined to make higher-level abstractions. (an Enemy object is an abstraction; a collection Enemy objects is a higher-level of abstraction)

Examples

Examples of abstractions:

1) a function/method/procedure that automates a behavior is an abstraction of that behavior.
   - arcade.draw_line(20,30,50,60) draws a line connecting (20,30) and (50,60). A programmer might not know the details of how this method is implemented but can still use it.

2) a data structure, such as a list of objects, models a collection of objects grouped together to simulate a real situation is an abstraction.

Application Programming Interface(API)

An ***application programming interface*** **(API)** is a set of routines, protocols and tools for building software applications.

For example, Amazon or eBay APIs allow developers or programmers to use the existing retail infrastructure to create specialized web stores.

Google Maps APIs lets developers embed Google Maps on webpages using a JavaScript or Flash interface.

Client-code programmers: programmers who uses those APIs to develop applications do not need to know the implementation details of the code.

Accessing Attributes/Data

Encapsulation stipulates that data inside the object should only be accessed through a public ***interface*** – that is, the object's methods.

```
# class definitions
class Circle:
      def __init__(self, diameter)
            self.diameter = diameter
            import math
            self.circumference = math.pi * self.diameter
# it is not "best practice" to directly access attributes outside
# of a class:
c = Circle(10)
print(c.diameter)
```

This is discouraged. Use an **accessor method** to access attributes.

Accessor Methods

Accessor methods allow client-code outside the class to access attributes. Most accessor are methods simply returns the value.

```
# class definitions
class Circle:
      def __init__(self, diameter)
            self.diameter = diameter
            import math
            self.circumference = math.pi * self.diameter
      def get_diameter(self):
            return self.diameter
c = Circle(10)
print(c.get_diameter())
```

accessor method

Calling an accessor method.

Mutator Methods

Mutator methods allow client-code outside the class to modify/mutate attributes. Directly modifying an attribute is not considered "best practice" and may lead to subtle errors.

What is the wrong with this code?

```
class Circle:
      def __init__(self, diameter)
            self.diameter = diameter
            import math
            self.circumference = math.pi * self.diameter
      def get_diameter(self):
            return self.diameter
c = Circle(10)
c.diameter = 20
```

This client code modifies an attribute directly and forgets to update the other attribute "circumference".

Mutator Methods

Mutator methods allow client-code outside the class to **modify/mutate** attributes.

```
import math
class Circle:
        def __init__(self, diameter)
                self.diameter = diameter
                self.circumference = math.pi * self.diameter
        def get_diameter(self):
                return self.diameter
        def set_diameter(self, new_diameter):
                self.diameter = new_diameter
                self.circumference = math.pi * self.diameter
```

mutator method

Inheritance

Inheritance gives us the ability to create a new class based on an existing class.

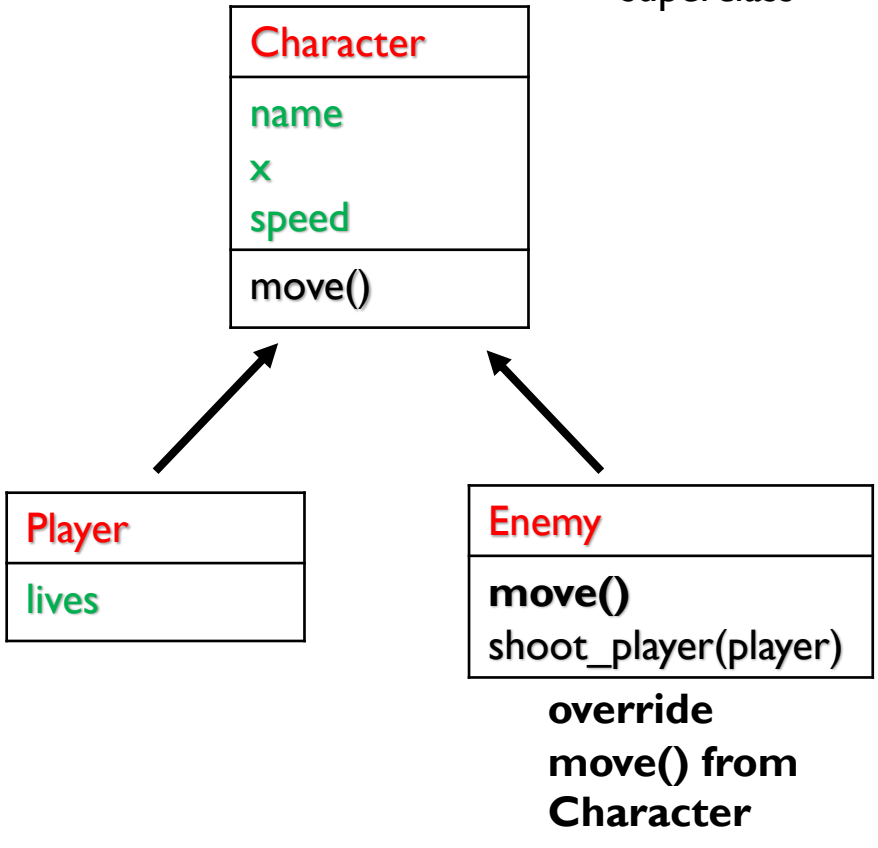
Inheritance allows us to **reuse** the code from our existing class.

For example, we can extend the Character class to another class: the Player class. The Character class is the **base class**, the **parent class** or the **superclass.** The Player class is a **subclass or child class** of Character.

We can also create the Enemy class which also inherits from Character.

Class Diagram

parent class
superclass



The subclasses Player and Enemy inherit all of the variables and methods from the Character class.

In this way, the Character class's code can be reused and built upon.

child class
subclass

game.py

```
class Character:
      def __init__(self, name, x, speed):
            self.name = name
            self.x = x
            self.speed = speed
      def move(self):
            self.x += self.speed

class Player(Character):
      def __init__(self, name, x, speed, lives):
            super().__init__(name, x, speed)
            self.lives = lives
```

The Player class inherits all of the variables and methods from Character(name, x, speed, move()).
However, it still needs to initialize them.

This calls the init method from the parent class to initialize its variables.

Player also needs to also initialize any new variables.

game.py

```
class Enemy(Character):
      def __init__(self, name, x, speed, player):
            super().__init__(name, x, speed)
            self.player = player
      # overrides move() from Character
      def move(self):
            super().move()
            …
            self.shoot_player(self.player)
            # remaining implementation not shown
      def shoot_player(self, player):
            # implementation not shown
```

This reuses move() code from parent class Character then add more code specific to how an Enemy moves. For example, an Enemy object moves and shoots at each frame.

continue on the next slide…

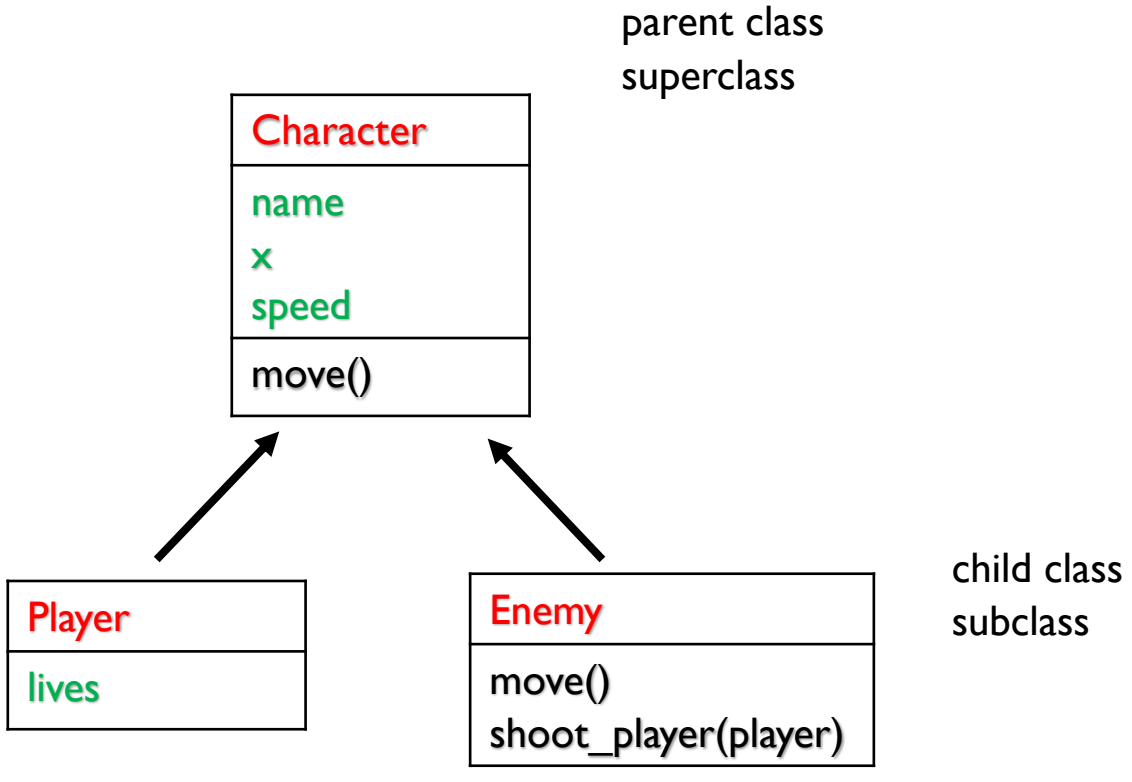
game.py(continue from last slide)

```
def main():
      p1 = Player(“Jack”, 10, 4, 3)
      e1 = Enemy(“Boss”, 20, -3)
      p1.move()      # move() inherited from parent class
                     # Character
      e1.move()      # move() is overridden by child class
                     # Enemy
      e1.shoot_player(p1)


main()
```

Is-A Relationship

A Player **is a** Character.
An Enemy **is a** Character.



isinstance

The built-in `isinstance(a, b)` function returns whether a is an instance or subclass of b.

```
In [1]: a = [0, 5, 2]
In [2]: isinstance(a, list)
Out [2]: True
In [3]: isinstance(a, str)
Out [3]: False
In [4]: b = "hi"
In [5]: isinstance(b, str)
Out [5]: True
```

isinstance

The built-in `isinstance(a, b)` function returns whether a is an instance or subclass of b.

Assume that Player is a subclass of Character as in the previous slides.

```
In[1]: p = Player("Mario", x=100, speed=5, lives=3)
In [2]: isinstance(p, Player)
Out [2]: True
In [2]: isinstance(p, Character)
Out [2]: True  # since Player is a subclass of Character
```

Inheritance

Inheritance is a powerful feature that allows us to inherit code from another class or library.

The **Arcade library**, for example, contains many classes and functions that we can use to write arcade games. It has a class called **Window**.

The **Window** class can create a window, draw shapes, images and animate them, detect and respond to keyboard and mouse inputs.

That's a lot of code that we don't want to write! Instead, we can simply inherit from it!

Inheritance

We first need to **import** the arcade library then declare the class that inherits from Window.

```
import arcade


class MyGame(arcade.Window):
      # MyGame inherits variables and methods from Window!!!
      # including the ability to animate images, detect
      # keyboard and mouse inputs, etc…
```

Lab 1

Modify the previous lecture's lab. In the Student class, add an accessor and a mutator method for each variable(name and gpa). These methods simply return/modify the variables.

Write the class GradStudent which inherits from the Student class. This class has an additional variable: researchTopic. Add accessor/mutator for researchTopic. Both the Student and GradStudent classes(and also average_gpa) should be in a module called objects.py.

Write the main method(in main.py) and:

1) Create a GradStudent object and store it in a variable. Print out name, gpa and researchTopic of the GradStudent object using the dot notation.
2) Create a list of three GradStudent objects.
3) Call average_gpa on the list of GradStudent objects.

Lab 2

Create a new repl.

Write three classes in a module called objects.py: Rectangle, Square and Cube. Square is a subclass of Rectangle and Cube is a subclass of Square.

A Rectangle object has length and width as attributes and accessor methods: area() and perimeter().

A Square object is a Rectangle object and has no additional attributes other than the ones inherited from Rectangle. It also inherits area() and perimeter() from Rectangle.

A Cube object is a Square object. It has no additional attributes but has two additional methods: area() which overrides Square's area() and volume(). Remember to call super().area() when implementing these.

Lab 2

Write the main method(main.py) and:

1) Create a Rectangle, Square and Cube object. Make sure calls to ALL of the methods for ALL of the objects work properly. Make 6 calls, 2 per object.
2) Use isinstance(a,b) to check relationships.

3) Put those objects in a list. Use a for loop to call area on each object. What do you notice? If you don't know the ordering of the objects in the list, it is unclear which area() is called and when. Python resolves this at runtime. This is called **polymorphism**, which means "many forms". Do you see why?

References

1) Halterman, Richard. Fundamentals of Python Programming. Southern Adventist University.