

香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen



SCHOOL OF
DATA SCIENCE
數據科學學院

Introduction to Computer Science: Programming Methodology

Lecture 7 Object Oriented Programming II

Tongxin Li

School of Data Science

Inheritance

- The object-oriented programming couples data and methods together into objects
- The object-oriented approach combines the power of the structural programming with an added dimension that integrates data with operations into objects
- Object-oriented programming (OOP) allows you to define new classes from existing classes. This is called inheritance
- Inheritance extends the power of the object-oriented paradigm by adding an important and powerful feature for reusing software

Superclass and subclass

- Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses)

- You use a class to model objects of the same type. Different classes may have some common properties and behaviours that you can generalize in a class

- Inheritance enables you to define a general class and later extend it to define more specialized classes

- The specialized classes inherit the properties and methods from the general class.

Inherited color methods

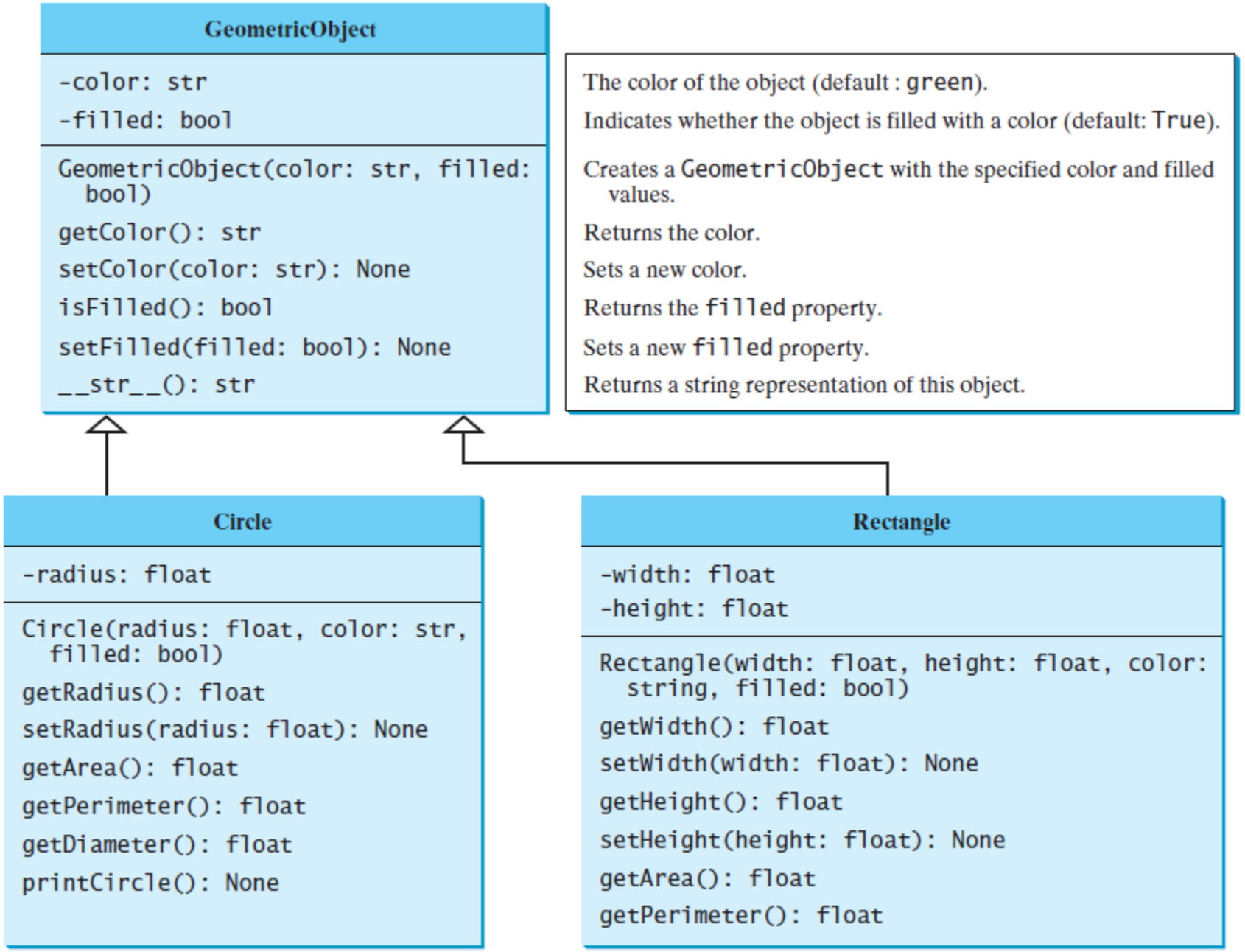
A circle and a rectangle both start green. What will this code print?

```
circle = Circle(2)
rectangle = Rectangle(3, 4)
circle.setColor("red")
print(circle.getColor(), rectangle.getColor())
```

Output `red green`

Circle and Rectangle inherit the color methods from GeometricObject.
Each object stores its own color.

Geometric Object and two of its subclasses



The code for GeometricObject

```
class GeometricObject:                                          GeometricObject class
    def __init__(self, color = "green", filled = True):         initializer
        self.__color = color                                    data fields
        self.__filled = filled

    def getColor(self):                                         getColor
        return self.__color

    def setColor(self, color):                                  setColor
        self.__color = color

    def isFilled(self):                                         isFilled
        return self.__filled

    def setFilled(self, filled):
        self.__filled = filled

    def __str__(self):
        return "color: " + self.__color + \
            " and filled: " + str(self.__filled)
```

The code for Circle class

- A subclass inherits accessible data fields and methods from its superclass, but it can also have other data fields and methods

In Python, the **super()** function is used to refer to the parent class or superclass. It allows you to call methods defined in the superclass from the subclass, enabling you to extend and customize the functionality inherited from the parent class.

```
from GeometricObject import GeometricObject
import math # math.pi is used in the class

class Circle(GeometricObject):
    def __init__(self, radius):
        super().__init__()
        self.__radius = radius

    def getRadius(self):
        return self.__radius

    def setRadius(self, radius):
        self.__radius = radius

    def getArea(self):
        return self.__radius * self.__radius * math.pi

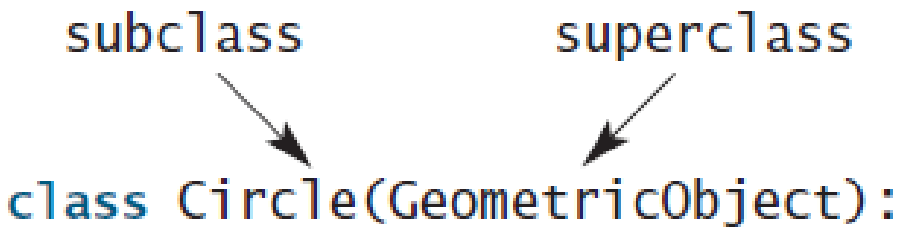
    def getDiameter(self):
        return 2 * self.__radius

    def getPerimeter(self):
        return 2 * self.__radius * math.pi

    def printCircle(self):
        print(self.__str__() + " radius: " + str(self.__radius))
```

Inheritance syntax

- The Circle class is derived from the GeometricObject class, based on the following syntax



- Circle class inherits the GeometricObject class, thus inheriting the methods getColor, setColor, isFilled, setFilled, and __str__
- The printCircle method invokes the __str__() method defined to obtain properties defined in the superclass

The code for rectangle class

```
from GeometricObject import GeometricObject

class Rectangle(GeometricObject):
    def __init__(self, width = 1, height = 1):
        super().__init__()
        self.__width = width
        self.__height = height

    def getWidth(self):
        return self.__width

    def setWidth(self, width):
        self.__width = width

    def getHeight(self):
        return self.__height

    def setHeight(self, height):
        self.__height = self.__height

    def getArea(self):
        return self.__width * self.__height

    def getPerimeter(self):
        return 2 * (self.__width + self.__height)
```

extend superclass

initializer

superclass initializer

methods

The code for testing Circle and Rectangle

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    circle = Circle(1.5)
    print("A circle", circle)
    print("The radius is", circle.getRadius())
    print("The area is", circle.getArea())
    print("The diameter is", circle.getDiameter())

    rectangle = Rectangle(2, 4)
    print("\nA rectangle", rectangle)
    print("The area is", rectangle.getArea())
    print("The perimeter is", rectangle.getPerimeter())

main() # Call the main function
```

```
A circle color: green and filled: True
The radius is 1.5
The area is 7.06858347058
The diameter is 3.0

A rectangle color: green and filled: True
The area is 8
The perimeter is 12
```

Remarks on super and sub-class

- A subclass is not a subset of its superclass; In fact, a subclass usually contains more information and methods than its superclass

- Inheritance models the **is-a** relationships, but not all **is-a** relationships should be modelled using inheritance

- Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship

Practice

```
class A:
    def __init__(self, i = 0):
        self.i = i

class B(A):
    def __init__(self, j = 0):
        self.j = j

def main():
    b = B()
    print(b.i)
    print(b.j)

main() # Call the main function
```

What is the problem with the above code?

Practice

What is the output of the following program?

```
class A:
    def __init__(self, i = 0):
        self.i = i

    def m1(self):
        self.i += 1

class B(A):
    def __init__(self, j = 0):
        super().__init__(3)
        self.j = j

    def m1(self):
        self.i += 1

def main():
    b = B()
    b.m1()
    print(b.i)
    print(b.j)

main() # Call the main function
```

Overriding methods

- A subclass inherits methods from a superclass

- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding

Example

```
class Animal:
    def make_sound(self):
        # Parent's default way
        print("Generic animal sound")

class Dog(Animal):
    def make_sound(self):
        # Dog's specific way (overrides parent)
        print("Woof! Woof!")

class Cat(Animal):
    def make_sound(self):
        # Cat's specific way (overrides parent)
        print("Meow.")

# --- Test it ---
my_animal = Animal()
my_dog = Dog()
my_cat = Cat()

my_animal.make_sound()  # Generic animal sound
my_dog.make_sound()     # Woof! Woof!
my_cat.make_sound()     # Meow.
```

The object class

- Every class in Python is descended from the object class

- The object class is defined in the Python library. If no inheritance is specified when a class is defined, its superclass is object by default



Methods of the object class

- The __new__() method is automatically invoked when an object is constructed. This method then invokes the __init__() method to initialize the object. Normally you should only override the __init__() method to initialize the data fields defined in the new class

- The __str__() method returns a string description for the object
- Often, we override the __str__() method so that it returns an informative description for the object
- The __eq__() method returns True if two objects are the same

What are "Dunder" Methods?

Dunder = **D**ouble **Under**score

They are special "hook" methods that you don't call directly.

Instead, Python calls them for you in response to specific syntax.

Examples:

`obj = MyClass()` triggers `__init__`

`print(obj)` triggers `__str__`

`obj1 + obj2` triggers `__add__`

`obj['key']` triggers `__getitem__`

Dunder Methods vs Regular Methods

Dunder methods are just like regular methods.

This means they follow the exact same rules of inheritance:

1. They are inherited from the parent class.

2. They can be overridden (replaced) by the child class.

3. They can be extended using super().

The *only* difference is **how** they are called.

Rule 1: Dunder Methods are Inherited

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        # This provides a default string for all Animals
        return f"An animal named {self.name}"

class Dog(Animal):
    # This class has NO __str__ method
    def bark(self):
        print("Woof!")

# --- Execution ---
d = Dog("Buddy")
print(d)  # Python looks for __str__ in Dog...
          # ...doesn't find it...
          # ...so it uses the one from Animal.
```

```
An animal named Buddy
```

Rule 2: Dunder Methods are Overridden

```
class Animal:
    def __str__(self):
        return f"An animal named {self.name}"

class Dog(Animal):
    # We OVERRIDE the __str__ method
    def __str__(self):
        return f"A Dog named {self.name}"

# --- Execution ---
d = Dog("Buddy")
print(d)  # Python looks for __str__ in Dog...
          # ...FINDS IT!
          # ...and runs it.
```

```
A Dog named Buddy
```

Rule 3: Dunder Methods are Extended with super()

```python
class Animal:
    def __init__(self, name):
        print("Animal __init__ running")
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        print("Dog __init__ running")

        # 1. Call the parent's __init__ to handle the 'name'
        super().__init__(name)

        # 2. Add new functionality for the Dog
        self.breed = breed

# --- Execution ---
d = Dog("Buddy", "Golden Retriever")
print(f"Name: {d.name}, Breed: {d.breed}")
```

```
Dog __init__ running
Animal __init__ running
Name: Buddy, Breed: Golden Retriever
```

Takeaways

No Magic: Dunder methods are not magic. They are just regular class methods that follow all the **normal rules** of inheritance (MRO later, super(), etc.):

1. Inherit by Default: Your class inherits all dunders from its parent (object is the default).

2. Override to Customize: You override dunders to make your class work cleanly with Python's syntax (like print, +, or len()).

3. Extend with super(): Use super() to add functionality to a parent's dunder method instead of replacing it (e.g., __init__).

Example

```
class GeometricObject:
    def __init__(self, color = "green", filled = True):
        self.__color = color
        self.__filled = filled

    def getColor(self):
        return self.__color

    def setFilled(self, filled):
        self.__filled = filled

    def __str__(self):
        return "color: " + self.__color + \
            " and filled: " + str(self.__filled)

class Circle(GeometricObject):
    # Other methods are omitted

    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

GeometricObject class

initializer

data fields

getColor

setColor

isFilled

__str__ in superclass

Example

- The __str__ method in the GeometricObject class returns the string describing a geometric object. This method can be overridden to return the string describing a circle

- The __str__() method is defined in the GeometricObject class and modified in the Circle class. Both methods can be used in the Circle class. To invoke the __str__ method defined in the GeometricObject class from the Circle class, use super().__str__()

```
class Circle(GeometricObject):
    # Other methods are omitted

    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

__str__ in superclass

Example

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties

    c = Circle(4)
    r = Rectangle(1, 3)
    displayObject(c)
    displayObject(r)
    print("Are the circle and rectangle the same size?",
        isSameArea(c, r))

# Display geometric object properties
def displayObject(g) :
    print(g.__str__())

# Compare the areas of two geometric objects
def isSameArea(g1, g2) :
    return g1.getArea() == g2.getArea()

main() # Call the main function
```

Output

```
color: green and filled: True radius: 4
color: green and filled: True width: 1 height: 3
Are the circle and rectangle the same size? False
```

Question

- Suppose you want to modify the displayObject function in previous example to perform the following tasks:

  ■ Display the area and perimeter of a GeometricObject instance

  ■ Display the diameter if the instance is a Circle, and the width and height if the instance is a Rectangle

```
# Display geometric object properties
def displayObject(g) :
    print(g.__str__())
```

Does this program work?

```
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())
    print("Diameter is", g.getDiameter())
    print("Width is", g.getWidth())
    print("Height is", g.getHeight())
```

Isinstance() function

- The isinstance() function can be used to determine whether an object is an instance of a class

- This function determines whether an object is an instance of a class by using the following syntax

```
isinstance(object, ClassName)
```

```
from CircleFromGeometricObject import Circle
from RectangleFromGeometricObject import Rectangle

def main():
    # Display circle and rectangle properties
    c = Circle(4)
    r = Rectangle(1, 3)
    print("Circle...")
    displayObject(c)
    print("Rectangle...")
    displayObject(r)

# Display geometric object properties
def displayObject(g):
    print("Area is", g.getArea())
    print("Perimeter is", g.getPerimeter())

    if isinstance(g, Circle):
        print("Diameter is", g.getDiameter())
    elif isinstance(g, Rectangle):
        print("Width is", g.getWidth())
        print("Height is", g.getHeight())

main() # Call the main function
```

```
Circle...
Area is 50.26548245743669
Perimeter is 25.132741228718345
Diameter is 8
Rectangle...
Area is 3
Perimeter is 8
Width is 1
Height is 3
```

Practice

What is the output of the following program?

```
class A:
    def __init__(self, i = 0):
        self.i = i

    def m1(self):
        self.i += 1

    def __str__(self):
        return 'The content of this object is:'+str(self.i)

x = A(8)
print(x)
```

Practice

What is the output of the following program?

```
class A:
    def __new__(self):
        print("A's __new__() invoked")

    def __init__(self):
        print("A's __init__() invoked"
```

```
class B(A):
    def __new__(self):
        print("B's __new__() invoked")

    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main() # Call the main function
```

Practice

What is the output of the following program?

```
class A:
    def __new__(self):
        self.__init__(self)
        print("A's __new__() invoked")

    def __init__(self):
        print("A's __init__() invoked")

class B(A):
    def __new__(self):
        self.__init__(self)
        print("B's __new__() invoked")

    def __init__(self):
        print("B's __init__() invoked")

def main():
    b = B()
    a = A()

main() # Call the main function
```

Polymorphism and dynamic binding

- Polymorphism means that an object of a subclass can be passed to a parameter of a superclass type
- A method may be implemented in several classes along the inheritance chain
- Python decides which method is invoked at runtime. This is known as dynamic binding

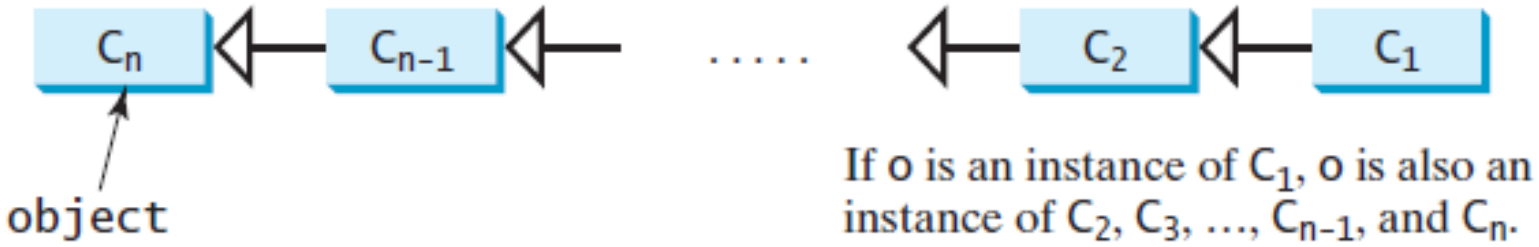
Meaning of Polymorphism: the ability of something to have or to be displayed in more than one form

Polymorphism

- **Reusability & Extension.** A subclass (child) inherits all features (attributes/methods) from its superclass (parent). The subclass can then add new features or override existing ones.
- **The "IS-A" Relationship.** A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
- Therefore, you can always use an instance of a subclass as its superclass type.

Dynamic binding

- Dynamic binding works as follows: Suppose an object o is an instance of classes C1, C2, ..., Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, ..., and Cn-1 is a subclass of Cn
- That is, Cn is the most general class, and C1 is the most specific class
- In Python, Cn is the **object** class
- If o invokes a method p, Python searches the implementation for the method p in C1, C2, ..., Cn-1, and Cn, in this order, until it is found



Example

- What would be the output of this program?

```
class C1:
    def __init__(self):
        self.f  = 1

    def output(self):
        print('In C1,  the f is:',self.f)

class C2(C1):
    def __init__(self):
        self.f  = 2

    def output(self):
        print('In C2,  the f is:',self.f)

class C3(C2):
    def __init__(self):
        self.f  = 3

class C4(C3):
    def __init__(self):
        self.f = 4

a=C4()
print(a.f)
a.output()
```

Example

```
class Student:
    def __str__(self):
        return "Student"

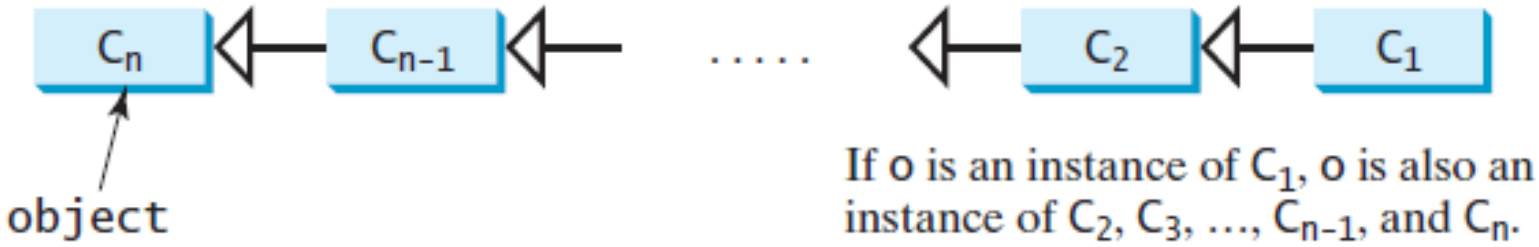

    def printStudent(self):
        print(self.__str__())

class GraduateStudent(Student):
    def __str__(self):
        return "Graduate Student"

a = Student()
b = GraduateStudent()
a.printStudent()
b.printStudent()
```

Data Hiding in Dynamic binding

- That is, Cn is the most general class, and C1 is the most specific class
- In Python, Cn is the **object** class
- If o invokes a method p, Python searches the implementation for the method p in C1, C2, ..., Cn-1, and Cn, in this order, until it is found
- Private variables and methods cannot be accessed



Practice

```
class Person:
    def getInfo(self):
        return "Person"

    def printPerson(self):
        print(self.getInfo())

class Student(Person):
    def getInfo(self):
        return "Student"

Person().printPerson()
Student().printPerson()
```

(a)

```
class Person:
    def __getInfo(self):
        return "Person"

    def printPerson(self):
        print(self.__getInfo())

class Student(Person):
    def __getInfo(self):
        return "Student"

Person().printPerson()
Student().printPerson()
```

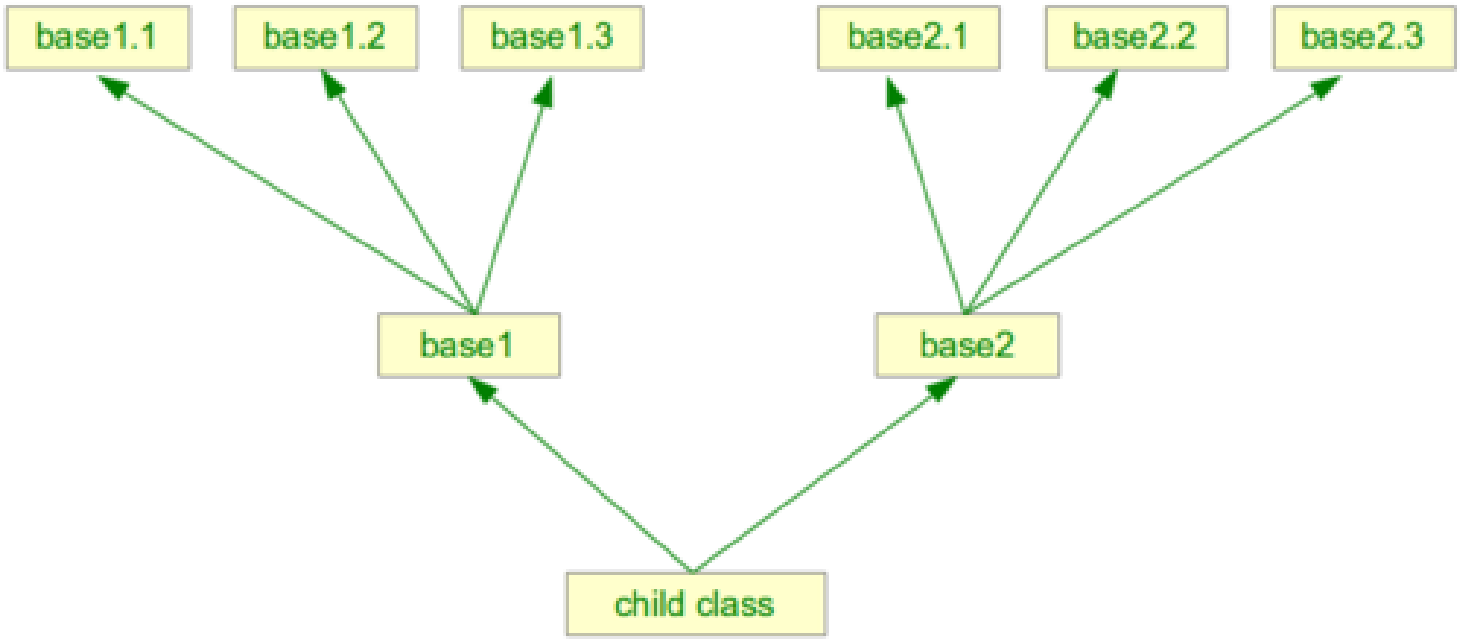
(b)

What are the outputs?

Multiple Inheritance

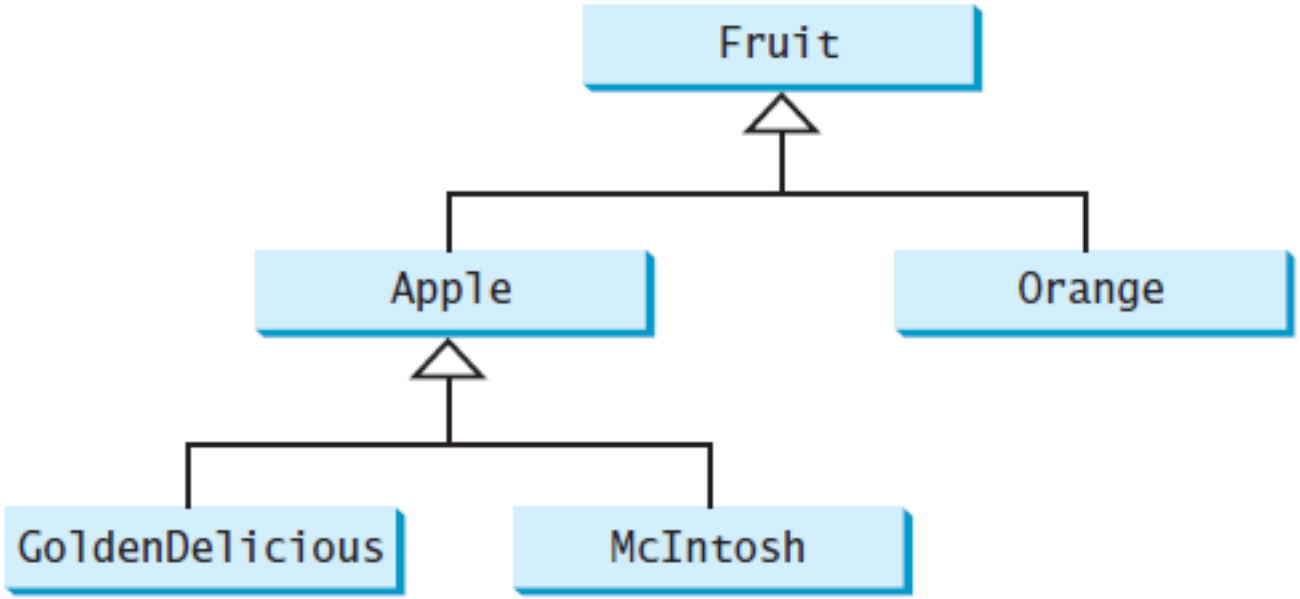
- In Python, we can define new class from multiple classes
- This is called multiple inheritance
- Multiple inheritance is a feature in which a class can inherit attributes and methods from more than one parent class

Inheritance Structure



- The inheritance relationship in Python can be represented by a graph structure

Practice



Assume that the following statements are given:

```
goldenDelicious = GoldenDelicious()
orange = Orange()
```

Questions

(a) Is goldenDelicious an instance of Fruit?

(b) Is goldenDelicious an instance of Orange?

(c) Is goldenDelicious an instance of Apple?

(d) Is goldenDelicious an instance of GoldenDelicious?

(e) Is goldenDelicious an instance of McIntosh?

(f) Is orange an instance of Orange?

(g) Is orange an instance of Fruit?

(h) Is orange an instance of Apple?

(i) Suppose the method makeAppleCider is defined in the Apple class. Can goldenDelicious invoke this method? Can orange invoke this method?

(j) Suppose the method makeOrangeJuice is defined in the Orange class. Can orange invoke this method? Can goldenDelicious invoke this method?

Example with two super() calls

```
class A():
    def __init__(self, a=100):
        self.a = a

class B():
    def __init__(self, b=200):
        self.b = b
```

What happens

Both calls reach A, so self.a becomes 2. The program prints 2 and 3, then raises AttributeError when it reads self.b.

```
class C(A, B):
    def __init__(self, a, b, c=300):
        super().__init__(a)
        super().__init__(b)
        self.c = c

    def output(self):
        print(self.a)
        print(self.c)
        print(self.b)

def main():
    c = C(1, 2, 3)
    c.output()

main()
```

Example with direct calls to A and B

```
class A():
    def __init__(self, a=100):
        self.a = a

class B():
    def __init__(self, b=200):
        self.b = b
```

What happens

A sets self.a to 1 and B sets self.b to 2.
The program prints 1, 3, then 2.

```
class C(A, B):
    def __init__(self, a, b, c=300):
        A.__init__(self, a)
        B.__init__(self, b)
        self.c = c

    def output(self):
        print(self.a)
        print(self.c)
        print(self.b)

def main():
    c = C(1, 2, 3)
    c.output()

main()
```

Method Resolution Order (Optional)

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

# You can print the MRO directly:
print(D.__mro__)

# You can also use the .mro() method:
# print(D.mro())
```

Output

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>)
```

```
class A:
    def __init__(self):
        print("Running A.__init__")

class B(A):
    def __init__(self):
        print("Running B.__init__")
        super().__init__()

class C(A):
    def __init__(self):
        print("Running C.__init__")
        super().__init__()

class D(B, C):
    def __init__(self):
        print("Running D.__init__")
        super().__init__()

print("Creating D with super():")
d = D()
```

Output

```
Creating D with super():
Running D.__init__
Running B.__init__
Running C.__init__
Running A.__init__
```

```
class A:
    def __init__(self):
        print("Running A.__init__")

class B(A):
    def __init__(self):
        print("Running B.__init__")
        A.__init__(self)  # Problem is here

class C(A):
    def __init__(self):
        print("Running C.__init__")
        A.__init__(self)  # Problem is here

class D(B, C):
    def __init__(self):
        print("Running D.__init__")
        B.__init__(self)
        C.__init__(self)

print("\nCreating D with hard-coded calls:")
d = D()
```

Output

```
Creating D with hard-coded calls:
Running D.__init__
Running B.__init__
Running A.__init__    First time
Running C.__init__
Running A.__init__    Second time! (BUG)
```

Example

```
A.__init__(self, a)
B.__init__(self, b)
```

What it does

Calls A's __init__ directly.
Calls B's __init__ directly.

Result

Works. The object has all three fields self.a, self.b and self.c.

Output

```
1
3
2
```

```
super().__init__(a)
super().__init__(b)
```

What it does

Calls __init__ on the next class in the MRO (A). The second call reaches A again.

Result

Fails. A.__init__ runs twice.
B.__init__ never runs.

Output

```
2
3
AttributeError
```

Choosing an initializer call

1 Prefer super()

```
super().__init__(a)
```

Use it for ordinary inheritance,
or when classes pass initialization
along a chain using super().

Python supplies self.

2 Choose A explicitly

```
A.__init__(self, a)
```

Use it to call A directly.
This fits our separate A and B classes,
which do not call super().

You must pass self.

super() follows the inheritance order. A.__init__ chooses A.