



香港中文大學(深圳)

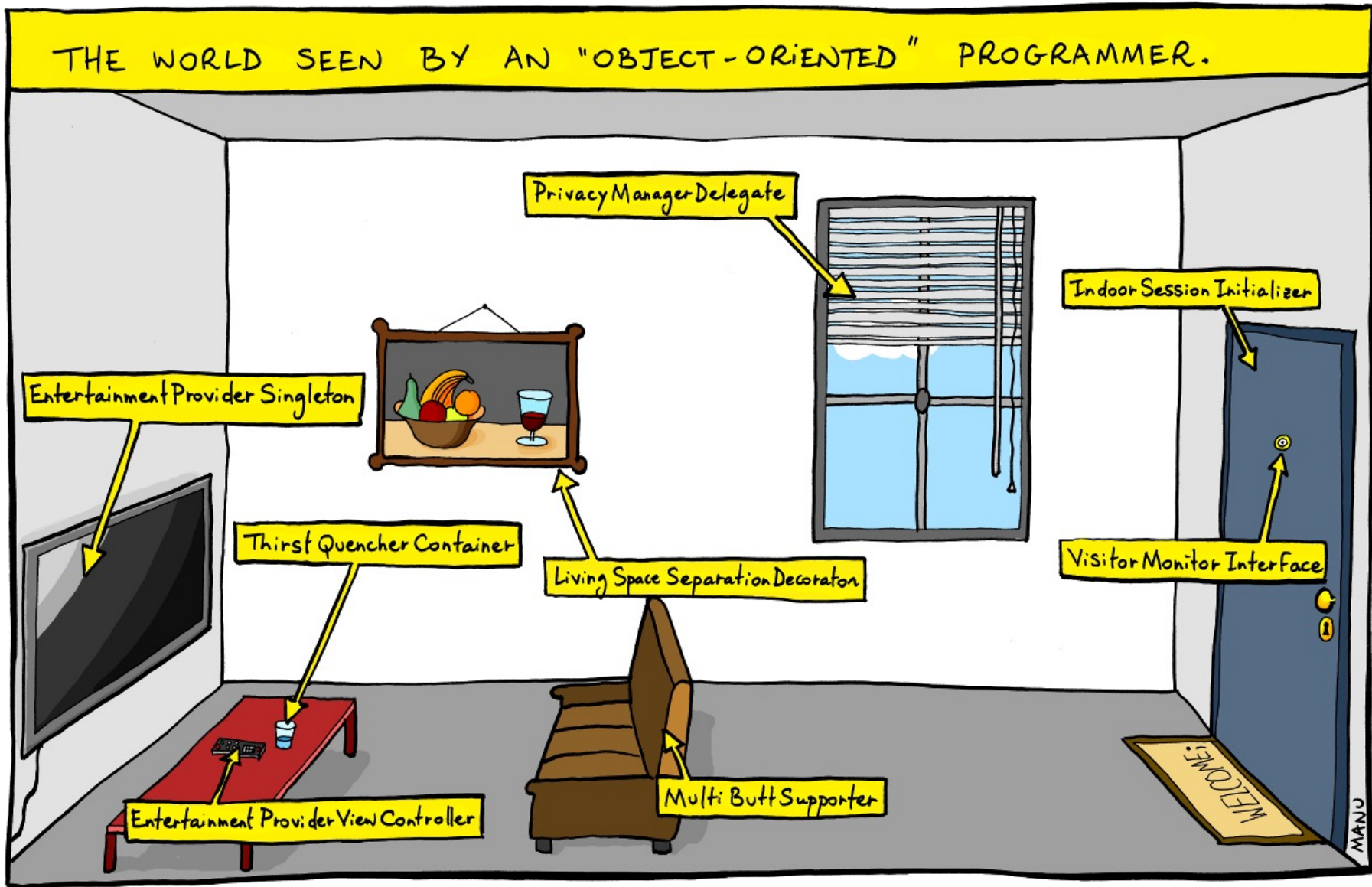
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 7 Object Oriented Programming II

Prof. Pinjia He

School of Data Science



OOP

- The object-oriented programming couples **data** and **methods** together into objects

```
class Circle:
    def __init__(self, radius=1, fillColor='black'):
        self.radius = radius
        self.fillColor = fillColor
    def getArea(self):
        return self.radius * self.radius * math.pi
```

OOP

- The object oriented approach combines the power of the **structural programming** with an added dimension that integrates data with operations into **objects**

If-else, loop, structures!!

Only objects!!



Geometric object

GeometricObject

-color: str

-filled: bool

GeometricObject(color: str, filled: bool)

getColor(): str

setColor(color: str): None

isFilled(): bool

setFilled(filled: bool): None

__str__(): str

The color of the object (default : green).

Indicates whether the object is filled with a color (default: True).

Creates a GeometricObject with the specified color and filled values.

Returns the color.

Sets a new color.

Returns the filled property.

Sets a new filled property.

Returns a string representation of this object.

The code for GeometricObject

<pre>class GeometricObject: def __init__(self, color = "green", filled = True): self.__color = color self.__filled = filled def getColor(self): return self.__color def setColor(self, color): self.__color = color def isFilled(self): return self.__filled def setFilled(self, filled): self.__filled = filled def __str__(self): return "color: " + self.__color + \ " and filled: " + str(self.__filled)</pre>	<p>GeometricObject class</p> <p>initializer</p> <p>data fields</p> <p>getColor</p> <p>setColor</p> <p>isFilled</p>
---	--

Circle and Rectangle

- We want to have **similar** data fields and methods **as GeometricObject**

```
-color: str  
-filled: bool
```

```
getColor(): str  
setColor(color: str): None  
isFilled(): bool  
setFilled(filled: bool): None  
__str__(): str
```

- We also want some **specialized** data fields and methods

```
-radius: float
```

```
getRadius(): float  
setRadius(radius: float): None  
getArea(): float  
getPerimeter(): float  
getDiameter(): float  
printCircle(): None
```

Inheritance

- 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.

Inheritance syntax

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

```
subclass      superclass
   \          /
    class Circle(GeometricObject):
```

- **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 Circle class

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

```
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))
```

Circle and Rectangle

Circle
-radius: float
Circle(radius: float, color: str, filled: bool)
getRadius(): float
setRadius(radius: float): None
getArea(): float
getPerimeter(): float
getDiameter(): float
printCircle(): None

Rectangle
-width: float -height: float
Rectangle(width: float, height: float, color: string, filled: bool)
getWidth(): float
setWidth(width: float): None
getHeight(): float
setHeight(height: float): None
getArea(): float
getPerimeter(): float

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
```

Some more information about 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?

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

- 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
```

Practice

What would be 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
```

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**

```
class ClassName:  
    ...
```

Equivalent

```
class ClassName(object):  
    ...
```

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
- Usually you should override the `__str__()` method so that it returns an informative description for the object

What is the output of this 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)
```

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
- Usually you should 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 is the output of this 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
```


What is the output of this 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

- The **inheritance** relationship enables a subclass to inherit features from its superclass with additional new features
- 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 pass an instance of a subclass to a **parameter** of its superclass type

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**

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()

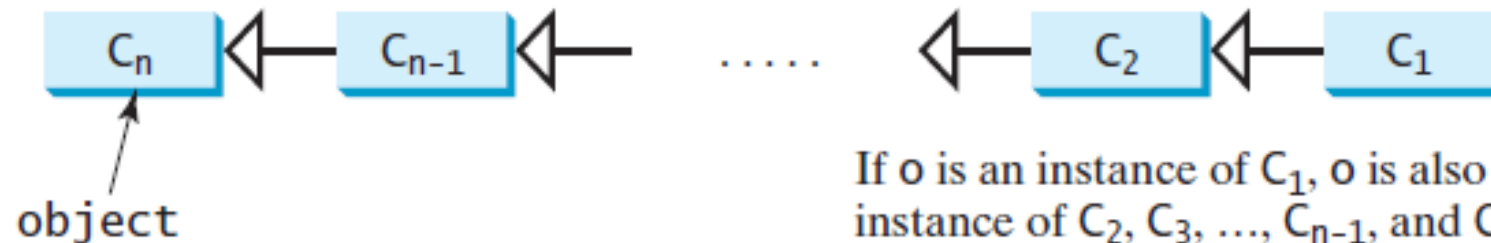
main() # Call the main function
```

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
```

Dynamic binding

- Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n , where C_1 is a subclass of C_2 , C_2 is a subclass of C_3 , ..., and C_{n-1} is a subclass of C_n
- That is, C_n is the most general class, and C_1 is the most specific class
- In Python, C_n is the object class
- If o invokes a method p , Python searches the implementation for the method p in C_1, C_2, \dots, C_{n-1} , and C_n , 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()
```


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

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

```
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())

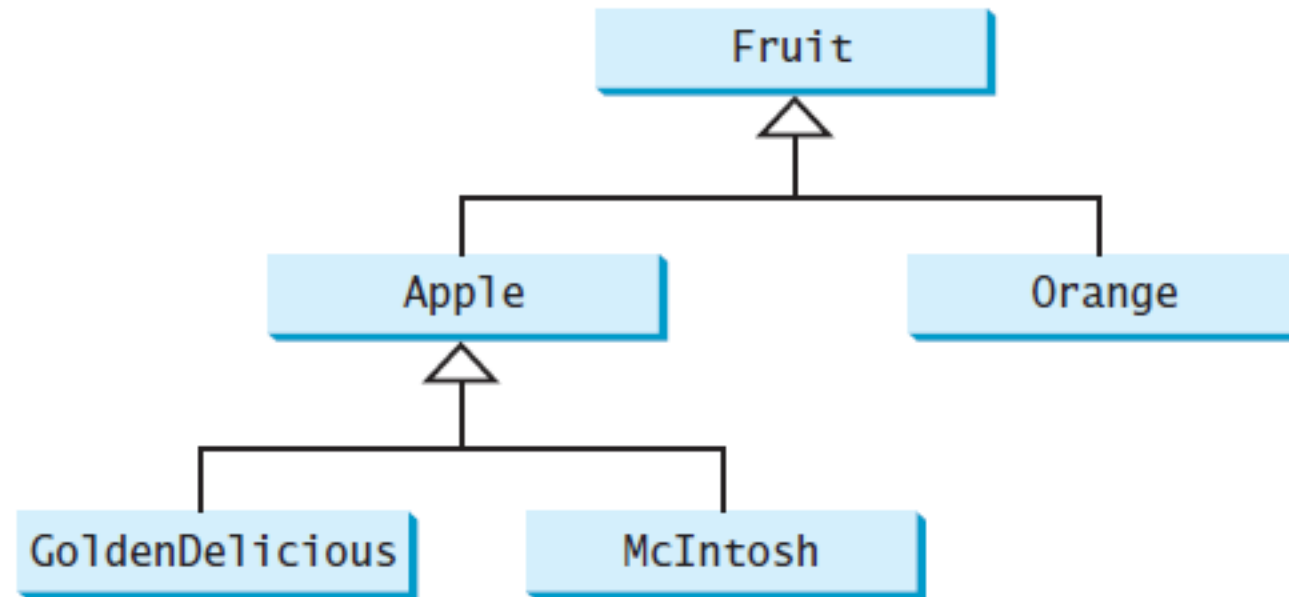
class Student(Person):
    def __getInfo(self):
        return "Student"

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

(b)

What would be the outputs?

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?

Practice: course class

Course
<code>-courseName: str</code> <code>-students: list</code>
<code>Course(courseName: str)</code> <code>getCourseName(): str</code> <code>addStudent(student: str): None</code> <code>dropStudent(student: str): None</code> <code>getStudents(): list</code> <code>getNumberOfStudents(): int</code>

The name of the course.

A list to store the students in the course.

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.

Answer

```
from Course import Course
```

```
def main():
```

```
    course1 = Course("Data Structures")  
    course2 = Course("Database Systems")
```

```
    course1.addStudent("Peter Jones")  
    course1.addStudent("Brian Smith")  
    course1.addStudent("Anne Kennedy")
```

```
    course2.addStudent("Peter Jones")  
    course2.addStudent("Steve Smith")
```

```
    print("Number of students in course1:",  
          course1.getNumberofStudents())  
    students = course1.getStudents()  
    for student in students:  
        print(student, end = ", ")
```

```
    print("\nNumber of students in course2:",  
          course2.getNumberofStudents())
```

```
main() # Call the main function
```

```
Number of students in course1: 3  
Peter Jones, Brian Smith, Anne Kennedy,  
Number of students in course2: 2
```

Answer

```
class Course:
    def __init__(self, courseName):
        self.__courseName = courseName
        self.__students = []

    def addStudent(self, student):
        self.__students.append(student)

    def getStudents(self):
        return self.__students

    def getNumberOfStudents(self):
        return len(self.__students)

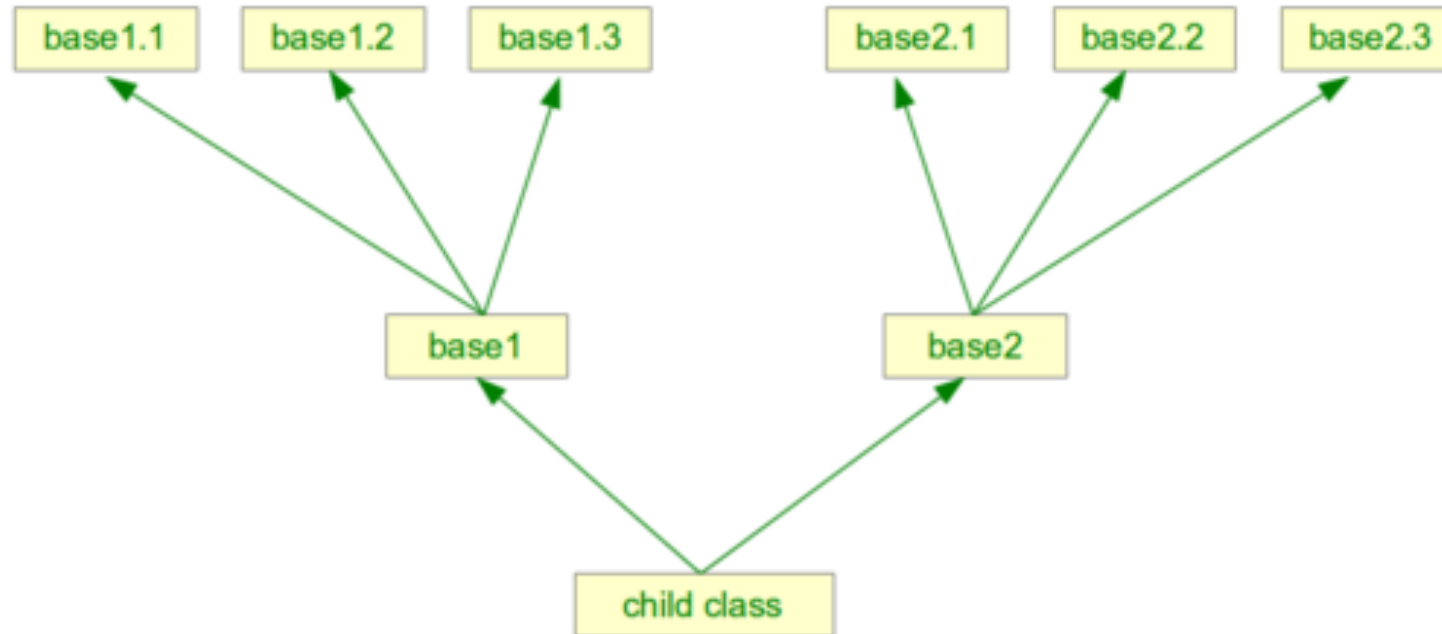
    def getCourseName(self):
        return self.__courseName

    def dropStudent(student):
        print("Left as an exercise")
```

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 Tree



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

Example

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

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

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

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

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()
```