

OOPs Features

Anoop Kumar Patel

Assistant Professor

Department of Computer Engineering

NIT Kurukshetra

Hiding Data Fields

- Making data fields private protects data and makes the class easy to maintain.
- Data fields can be accessed via instance variables directly from an object.
- i.e. the following code, provide access of the circle's radius from `c.radius`, is legal:

```
>>> c = Circle(5)
>>> c.radius = 5.4 # Access instance variable directly
>>> print(c.radius) # Access instance variable directly
5.4
>>>
```

- However, direct access of a data field in an object is not a good practice - for two reasons:
 - First, data may be tampered with. i.e., channel in the TV class has a value between 1 and 120, but it may be mistakenly set to an arbitrary value (e.g., `tv1.channel = 125`).
 - Second, the class becomes difficult to maintain and vulnerable to bugs. Let, we want to modify the Circle class to ensure that the radius is nonnegative after other programs have already used the class. For this we have to change not only the Circle class but also the programs that use it, because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).
- To prevent direct modifications of data fields, don't let the client directly access data fields. This is known as data hiding. This can be done by defining private data fields.

- In Python, the private data fields are defined with two leading underscores.
- We can also define a private method named with two leading underscores.
- Private data fields and methods can be accessed within a class, but they cannot be accessed outside the class.
- To make a data field accessible for the client, provide a get method to return its value.
- To enable a data field to be modified, provide a set method to set a new value.
- Colloquially, a get method is referred to as a getter (or accessor), and a set method is referred to as a setter (or mutator).

A **get** method has the following header:

```
def getPropertyname(self):
```

If the return type is Boolean, the **get** method is defined as follows by convention:

```
def isPropertyName(self):
```

A **set** method has the following header:

```
def setPropertyName(self, propertyValue):
```

CircleWithPrivateRadius.py

```
1  import math
2
3  class Circle:
4      # Construct a circle object
5      def __init__(self, radius = 1):
6          self.__radius = radius
7
```

```
8     def getRadius(self):
9         return self.__radius
10
11     def getPerimeter(self):
12         return 2 * self.__radius * math.pi
13
14     def getArea(self):
15         return self.__radius * self.__radius * math.pi
```

The radius property cannot be directly accessed in this new Circle class.

However, you can read it by using the getRadius() method. i.e.:

```
1 >>> from CircleWithPrivateRadius import Circle
2 >>> c = Circle(5)
3 >>> c.__radius
4 AttributeError: no attribute '__radius'
5 >>> c.getRadius()
6 5
7 >>>
```

Tip

If a class is designed for other programs to use, to prevent data from being tampered with and to make the class easy to maintain, define data fields as private. If a class is only used internally by your own program, there is no need to hide the data fields.

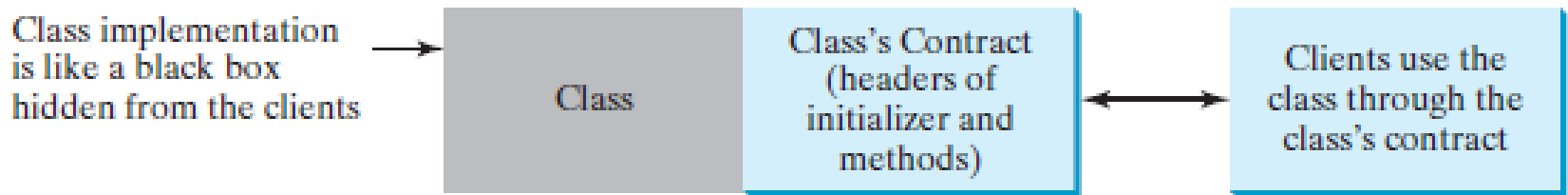
Note

Name private data fields and methods with two leading underscores, but don't end the name with more than one underscores. The names with two leading underscores and two ending underscores have special meaning in Python. For example, `__radius` is a private data field, but, `__radius__` is not a private data field.

Class Abstraction and Encapsulation

Class abstraction is a concept that separates class implementation from the use of a class. The class implementation details are invisible from the user.

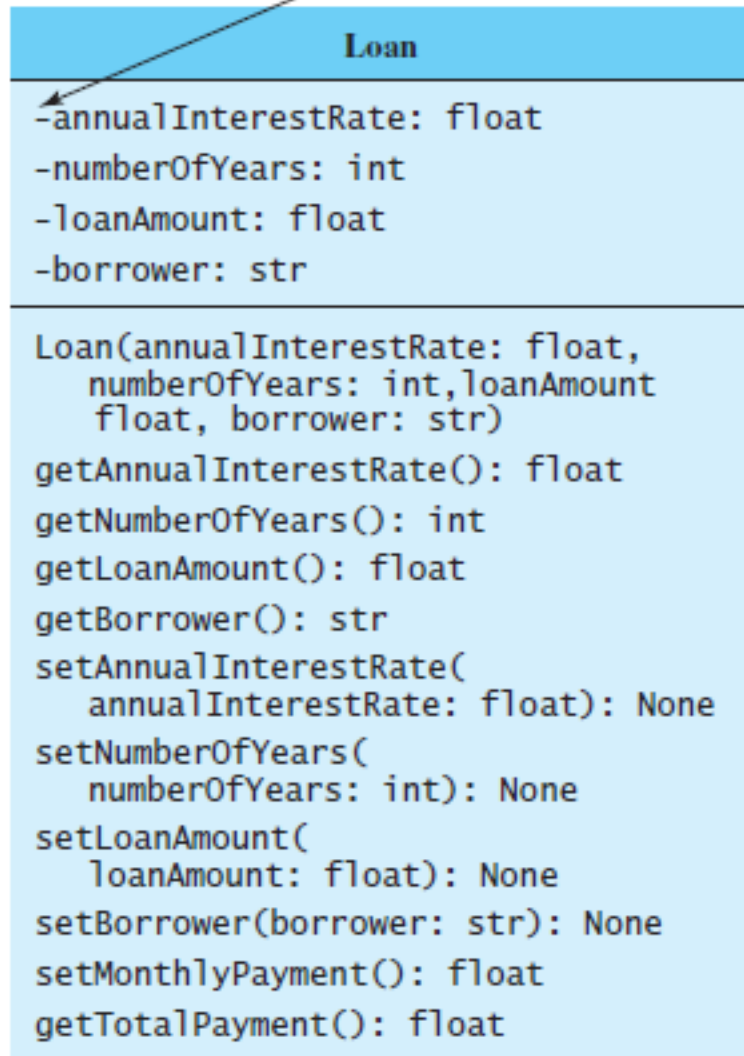
- The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.
- In essence, encapsulation combines data and methods into a single object and hides the data fields and method implementation from the user.
- A class is also known as an abstract data type (ADT).



Class abstraction separates class implementation from the use of the class.

- Let us consider getting a loan. A specific loan can be viewed as an object of a Loan class.
- The interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its methods.
- When we buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period.
- We can then use the methods to find the monthly payment and total payment of your loan.
- As a user of the Loan class, we don't need to know how these methods are implemented.

The - sign denotes a private data field.



The annual interest rate of the loan (default 2.5).

The number of years for the loan (default 1).

The loan amount (default 1000).

The borrower of this loan (default " ").

Constructs a **Loan** object with the specified annual interest rate, number of years, loan amount, and borrower.

Returns the annual interest rate of this loan.

Returns the number of the years of this loan.

Returns the amount of this loan.

Returns the borrower of this loan.

Sets a new annual interest rate for this loan.

Sets a new number of years for this loan.

Sets a new amount for this loan.

Sets a new borrower for this loan.

Returns the monthly payment of this loan.

Returns the total payment of this loan.

The UML diagram for the **Loan** class models (shows) the properties and behaviors of loans.

Important Pedagogical Tip

The UML diagram for the **Loan** class is shown in Figure 7.9. You should first write a test program that uses the **Loan** class even though you don't know how the **Loan** class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of the book.
- It is easier to learn how to implement a class if you are familiar with the class through using it.

For all the class development examples from now on, first create an object from the class and try to use its methods and then turn your attention to its implementation.

Inheritance

- 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.
- Suppose that we want to define classes to model circles, rectangles, and triangles.
- These classes have many common features.
- What is the best way to design these classes to avoid redundancy and make the system easy to comprehend and maintain?
- The answer is to use inheritance.

Superclasses and Subclasses

- Inheritance enables you to define a general class (a superclass) and later extend it to more specialized classes (subclasses).
- We use a class to model objects of the same type.
- Different classes may have some common properties and behaviors that you can generalize in a class, which can then be shared by other classes.
- 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.

Note

In OOP terminology, a class **C1** extended from another class **C2** is called a *derived class*, *child class*, or *subclass*, and **C2** is called a *base class*, *parent class*, or *superclass*. For consistency, this book uses the terms “subclass” and “superclass.”

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


In the following example:

- The Circle class inherits all accessible data fields and methods from the GeometricObject class. In addition, it has a new data field, radius, and its associated get and set methods. It also contains the getArea(), getPerimeter(), and getDiameter() methods for returning the area, perimeter, and diameter of a circle. The printCircle() method is defined to print the information about the circle.

of the rectangle.

- The Rectangle class inherits all accessible data fields and methods from the GeometricObject class. In addition, it has the data fields width and height and the associated get and set methods. It also contains the getArea() and getPerimeter() methods for returning the area and perimeter


subclass superclass



```
class Circle(GeometricObject):
```

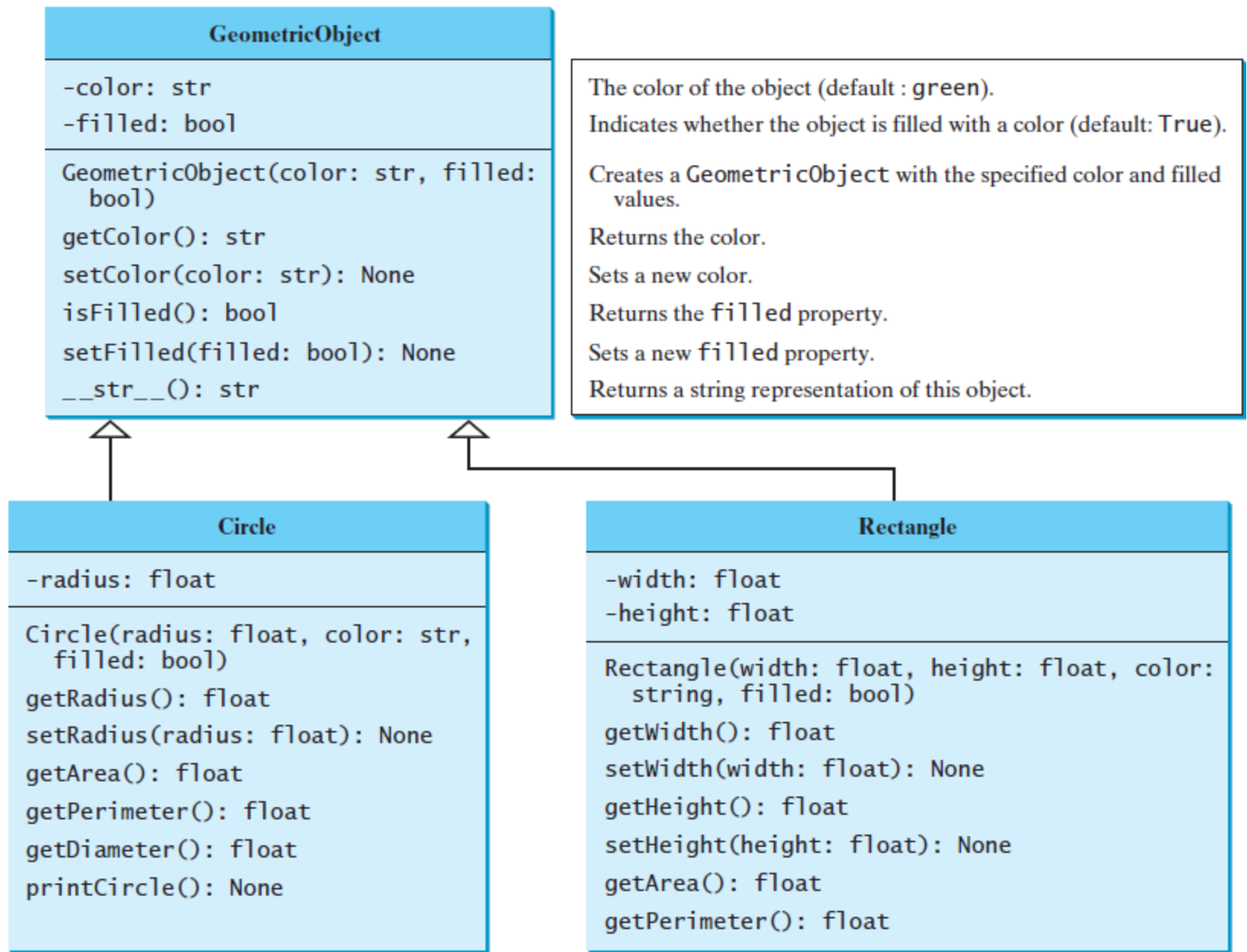
The diagram shows the word 'subclass' on the left and 'superclass' on the right. Two arrows point from these words to the word 'GeometricObject' in the code snippet below, which is part of the class definition for 'Circle'.

subclass superclass



```
class Rectangle(GeometricObject):
```

The diagram shows the word 'subclass' on the left and 'superclass' on the right. Two arrows point from these words to the word 'GeometricObject' in the code snippet below, which is part of the class definition for 'Rectangle'.



The **GeometricObject** class is the superclass for **Circle** and **Rectangle**

The following points regarding inheritance are worthwhile to note:

- Contrary to the conventional interpretation, 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 modeled using inheritance. For example, a square is a rectangle, but you should not extend a **Square** class from a **Rectangle** class, because the **width** and **height** properties are not appropriate for a square. Instead, you should define a **Square** class to extend the **GeometricObject** class and define the **side** property for the side of a square.
- 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.
- Python allows you to derive a subclass from several classes. This capability is known as *multiple inheritance*. To define a class derived from multiple classes, use the following syntax:

```
class Subclass(SuperClass1, SuperClass2, ...):  
    initializer  
    methods
```

Overriding Methods

- To override a method, the method must be defined in the subclass using the same header as in its superclass.
- 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.
- 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.
- To override it, add the following new method, `CircleFromGeometricObject.py`:

```
1 class Circle(GeometricObject):
2     # Other methods are omitted
3
4     # Override the __str__ method defined in GeometricObject
5     def __str__(self):
6         return super().__str__() + " radius: " + str(radius)
```

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__()` (line 6).

Similarly, you can override the `__str__` method in the `Rectangle` class as follows:

```
def __str__(self):
    return super().__str__() + " width: " + \
        str(self.__width) + " height: " + str(self.__height)
```

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. For example, the following two class definitions are the same:



- 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.
- By default, it returns a string consisting of a class name of which the object is an instance and the object's memory address in hexadecimal format.

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.
- 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.
- For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

PolymorphismDemo.py

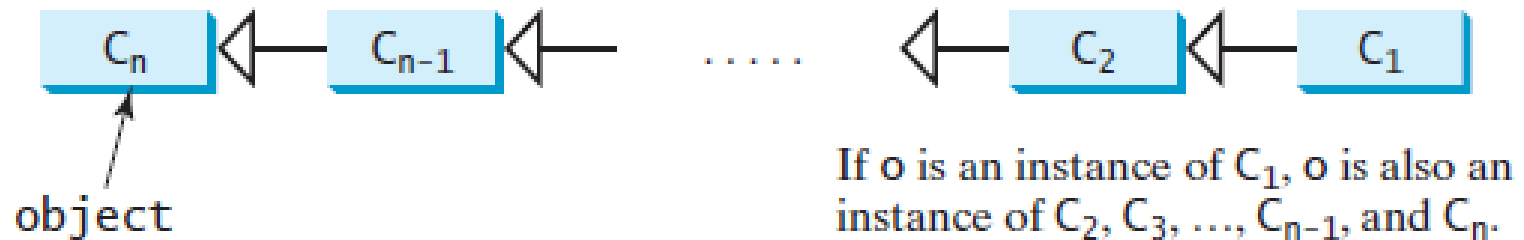
```
1  from CircleFromGeometricObject import Circle
2  from RectangleFromGeometricObject import Rectangle
3
4  def main():
5      # Display circle and rectangle properties
6      c = Circle(4)
7      r = Rectangle(1, 3)
8      displayObject(c)
9      displayObject(r)
10     print("Are the circle and rectangle the same size?",
11           isSameArea(c, r))
12
13     # Display geometric object properties
14     def displayObject(g):
15         print(g.__str__())
16
17     # Compare the areas of two geometric objects
18     def isSameArea(g1, g2):
19         return g1.getArea() == g2.getArea()
20
21     main() # Call the main function
```

```
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
```

- The `displayObject` method (line 14) takes a parameter of the `GeometricObject` type.
- You can invoke `displayObject` by passing any instance of `GeometricObject` (for example, `Circle(4)` and `Rectangle(1, 3)` in lines 8–9).
- An object of a subclass can be used wherever its superclass object is used. **This is commonly known as polymorphism** (from a Greek word meaning “many forms”).
- As seen in this example, `c` is an object of the `Circle` class. `Circle` is a subclass of `GeometricObject`. The `__str__()` method is defined in both classes.

- So, which `__str__()` method is invoked by `g` in the `displayObject` method (line 15)?
- The `__str__()` method invoked by `g` is determined using dynamic binding.
- Dynamic binding works as follows:
 Suppose an object `o` is an instance of classes `C 1` , `C 2` , ..., `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` , as shown in next slide.
- 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` , ..., `C n-1` , and `C n` , in this order, until it is found.

- Once an implementation is found, the search stops and the first-found implementation is invoked.



The method to be invoked is dynamically bound at runtime.

DynamicBindingDemo.py

```
1 class Student:
2     def __str__(self):
3         return "Student"
```

```
4
5     def printStudent(self):
6         print(self.__str__())
7
8     class GraduateStudent(Student):
9         def __str__(self):
10             return "Graduate Student"
11
12 a = Student()
13 b = GraduateStudent()
14 a.printStudent()
15 b.printStudent()
```

Student
Graduate Student

- Since a is an instance of Student, the printStudent method in the Student class is invoked for a.printStudent() (line 14), which invokes the Student class's __str__() method to return Student.
- No printStudent method is defined in GraduateStudent. However, since it is defined in the Student class and GraduateStudent is a subclass of Student, the printStudent method in the Student class is invoked for b.printStudent() (line 15).
- The printStudent method invokes GraduateStudent's __str__() method to display Graduate Student, since the object b that invokes printStudent is GraduateStudent (lines 6 and 10).