Classes and Inheritance
The items of a list can have any data type, even a class defined by the programmer.

Use append to insert an object into the list.

See RectangleList.py
What is inheritance?
What is inheritance?

- The idea behind inheritance is that a new class can be defined to borrow behavior from another class.
What is inheritance?

- The idea behind inheritance is that a new class can be defined to borrow behavior from another class.
- **Inheritance** is a feature of OOP that allows us to define a new class (called the *subclass, child class, or derived class*) that is a modified version of an existing class (called the *superclass, parent class, or base class*).
The subclass can:
The subclass can:

- Inherit properties, data, and methods of the superclass
The subclass can:

- Inherit properties, data, and methods of the superclass
- Add some of its own properties, data, and methods
The subclass can:

- Inherit properties, data, and methods of the superclass
- Add some of its own properties, data, and methods
- Override some of the superclass’ methods
What are the benefits of inheritance?
What are the benefits of inheritance?

- We can structure the classes of a system to avoid duplication of operations - i.e., we can make use of pre-existing methods.
What are the benefits of inheritance?

- We can structure the classes of a system to avoid duplication of operations - i.e., we can make use of pre-existing methods
- New classes can often be based on existing classes, promoting code reuse
The “is-a” Relationship

Child classes are specializations of their parent’s class. The child normally has all the characteristics of their parents, but more functionality. Such child classes have a so-called “is-a” relationship with their parent class.

Examples:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee</td>
<td>hourly employee, salaried employee</td>
</tr>
<tr>
<td>regular polygon</td>
<td>equilateral triangle, square</td>
</tr>
<tr>
<td>animal</td>
<td>mammal, bird, crustacean</td>
</tr>
</tbody>
</table>
Child classes are specializations of their parent’s class.
The “is-a” Relationship

- Child classes are specializations of their parent’s class
- The child normally has all the characteristics of their parents, but more functionality
Child classes are specializations of their parent’s class
The child normally has all the characteristics of their parents, but more functionality
Such child classes have a so-called “is-a” relationship with their parent class.
The “is-a” Relationship

- Child classes are specializations of their parent’s class
- The child normally has all the characteristics of their parents, but more functionality
- Such child classes have a so-called “is-a” relationship with their parent class.
- Examples:
The “is-a” Relationship

- Child classes are specializations of their parent’s class.
- The child normally has all the characteristics of their parents, but more functionality.
- Such child classes have a so-called “is-a” relationship with their parent class.
- Examples:

<table>
<thead>
<tr>
<th>Parent</th>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee</td>
<td>hourly employee, salaried employee</td>
</tr>
<tr>
<td>regular polygon</td>
<td>equilateral triangle, square</td>
</tr>
<tr>
<td>animal</td>
<td>mammal, bird, crustacean</td>
</tr>
</tbody>
</table>
Example 2 - Inheritance

See subclass.py

In the following class definitions, the statement

```python
class PFstudent(Student):
```

specifies that PFstudent is a subclass of the superclass Student and inherits all the properties and methods of the class Student.
In our previous example, notice that the subclass PFstudent and superclass Student both defined the method calcGrade. How did the program know which version to use?
In our previous example, notice that the subclass PFStudent and superclass Student both defined the method calcGrade. How did the program know which version to use?

- A subclass can change the behavior of an inherited method.
In our previous example, notice that the subclass PFStudent and superclass Student both defined the method calcGrade. How did the program know which version to use?

- A subclass can change the behavior of an inherited method.
- If a method defined in the subclass has the same name as a method in its superclass, the child’s method will override the parent’s method.
In our previous example, notice that the subclass PFStudent and superclass Student both defined the method calcGrade. How did the program know which version to use?

- A subclass can change the behavior of an inherited method.
- If a method defined in the subclass has the same name as a method in its superclass, the child’s method will override the parent’s method.
- This is an example of polymorphism.
The word **polymorphism** literally means “many forms.”
The word **polymorphism** literally means “many forms.”

When used in OOP, this refers to the fact that what an object does depends in response to a message (i.e. a method call) depends on the type/class of the object.
The word **polymorphism** literally means “many forms.”

When used in OOP, this refers to the fact that what an object does depends in response to a message (i.e. a method call) depends on the type/class of the object.

The feature of Python (and every OOP language) that allows two classes to use the same method name (but with different implementations) is called **polymorphism**.
Benefits of polymorphism?
Benefits of polymorphism?

- Gives OOP the flexibility for each object to perform an action just the way that it should be performed for that object.
Benefits of polymorphism?

- Gives OOP the flexibility for each object to perform an action just the way that it should be performed for that object.
- Allows the program to automatically adjust the implementation of the called method depending on the type of method that calls it.
Child classes can also add properties (i.e. instance variables). In order to do this, the child’s initializer must have the following parameter list:
Child classes can also add properties (i.e. instance variables). In order to do this, the child’s initializer must have the following parameter list:

- Begin with `self`
Child classes can also add properties (i.e. instance variables). In order to do this, the child’s initializer must have the following parameter list:

- Begin with `self`
- List the parent’s parameters
Child classes can also add properties (i.e. instance variables). In order to do this, the child’s initializer must have the following parameter list:

- Begin with `self`
- List the parent’s parameters
- Add on the new child’s parameters
Example:

class PFstudent(Student):
    def __init__(self, name, midterm, final, fullTime):
        # Import base parameters
        super().__init__(name, midterm, final)

        # Initialize the new parameters
        self.fullTime = fullTime
A statement of the form

```
isinstance(object, className)
```

returns True if object is of the named class (or any of its subclasses) and otherwise returns False.

The `isinstance` function can be applied to both built-in and user-defined classes.
The `isinstance` Function

Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isinstance('Hello', str)</code></td>
<td>True</td>
</tr>
<tr>
<td><code>isinstance(3.4, int)</code></td>
<td>False</td>
</tr>
<tr>
<td><code>isinstance(3.4, float)</code></td>
<td>True</td>
</tr>
<tr>
<td><code>isinstance([1, 2, 3], list)</code></td>
<td>True</td>
</tr>
<tr>
<td><code>isinstance([], list)</code></td>
<td>True</td>
</tr>
</tbody>
</table>

See also: `isinstance.py`
Encapsulation

Encapsulation is the process of packaging some data along with the set of operations that can be performed on the data. It is one of the major attractions of using objects. Encapsulation separates the actual implementation of an object from its use. The main program only has to worry about what objects can do, not how they are implemented. Encapsulation supports code reuse.

Encapsulation
Encapsulation

- is the process of packaging some data along with the set of operations that can be performed on the data.
Encapsulation

- is the process of packaging some data along with the set of operations that can be performed on the data.
- is one of the major attractions of using objects.
Encapsulation

- is the process of packaging some data along with the set of operations that can be performed on the data.
- is one of the major attractions of using objects.
- separates the actual implementation of an object from its use. The main program only has to worry about what objects can do, not how they are implemented.
Encapsulation

- is the process of packaging some data along with the set of operations that can be performed on the data.
- is one of the major attractions of using objects.
- separates the actual implementation of an object from its use. The main program only has to worry about what objects can do, not how they are implemented.
- supports code reuse.
Encapsulation is only a programming convention in Python - it is not enforced by the language. Strictly speaking, the accessor and mutator methods in our classes are not necessary. Example: Could have written

```python
s = Student("Elizabeth", 90, 85)
s.name = "Liz"
```
instead of

```python
s = Student("Elizabeth", 90, 85)
s.setName("Liz")
```
Encapsulation is only a programming *convention* in Python - it is not enforced by the language.
Encapsulation is only a programming convention in Python - it is not enforced by the language.

Strictly speaking, the accessor and mutator methods in our classes are not necessary.
Encapsulation is only a programming *convention* in Python - it is not enforced by the language.

Strictly speaking, the accessor and mutator methods in our classes are not necessary.

Example: Could have written

```python
s = Student("Elizabeth", 90, 85)
s.name = "Liz"
```

instead of

```python
s = Student("Elizabeth", 90, 85)
s.setName("Liz")
```
In Python, you can access the instance variables of any object with the regular dot notation.
In Python, you can access the instance variables of any object with the regular dot notation.

- Handy for testing
In Python, you can access the instance variables of any object with the regular dot notation.

- Handy for testing
- Poor practice in programs
In Python, you can access the instance variables of any object with the regular dot notation.

- Handy for testing
- Poor practice in programs
- References to instance variables should remain inside the class definition with the rest of the implementation details
In Python, you can access the instance variables of any object with the regular dot notation.

- Handy for testing
- Poor practice in programs
- References to instance variables should remain inside the class definition with the rest of the implementation details
- From outside the class, all interaction with an object should be done using the interface provided by its methods
Advantages of Encapsulation:
Advantages of Encapsulation:

- Allows us to modify and improve classes independently, without worrying about “breaking” other parts of the program.
Advantages of Encapsulation:

- Allows us to modify and improve classes independently, without worrying about “breaking” other parts of the program.
- As long as the interface provided by a class stays the same, the rest of the program can’t even tell that a class has changed.
See rpg.py