Classes and Inheritance

Lecture 24, Week 8
March 11, 2011
CSC108H1S
Velian Pandeliev

The `Database` Class

We will now write a `Database` class which can store a collection of `Movie`s. It should have:

- a list as an instance variable to store `Movie`s
- an `__init__` method
- a method called `populate` which reads comma-separated entries from a file in the form:

  `movie title,YYYY,genre`

- a `list_library` method which returns the list of movies sorted according to the `Movie`'s `__cmp__` function
- a `search` method which returns all movies whose titles match a search string

Sorting By Name

```
def __cmp__(self, other):
    '''Ordered by title'''
    if self.title > other.title:
        return 1
    elif self.title == other.title:
        return 0
    else:
        return -1
```

This `__cmp__` method sorts the `Movie` objects by name when we run `sort()`.

Sorting By Year

To sort our movies by name, we change the `__cmp__` method as follows:

```
def __cmp__(self, oth):
  '''Ordered by year'''
  if self.year > oth.year:
     return 1
  elif self.year == oth.year:
        return 0
  else:
        return -1
```

Relationships Between Classes

As the building blocks of more complex systems, objects can be designed to interact with each other in one of three ways:

Association: an object is aware of another object and holds a reference to it

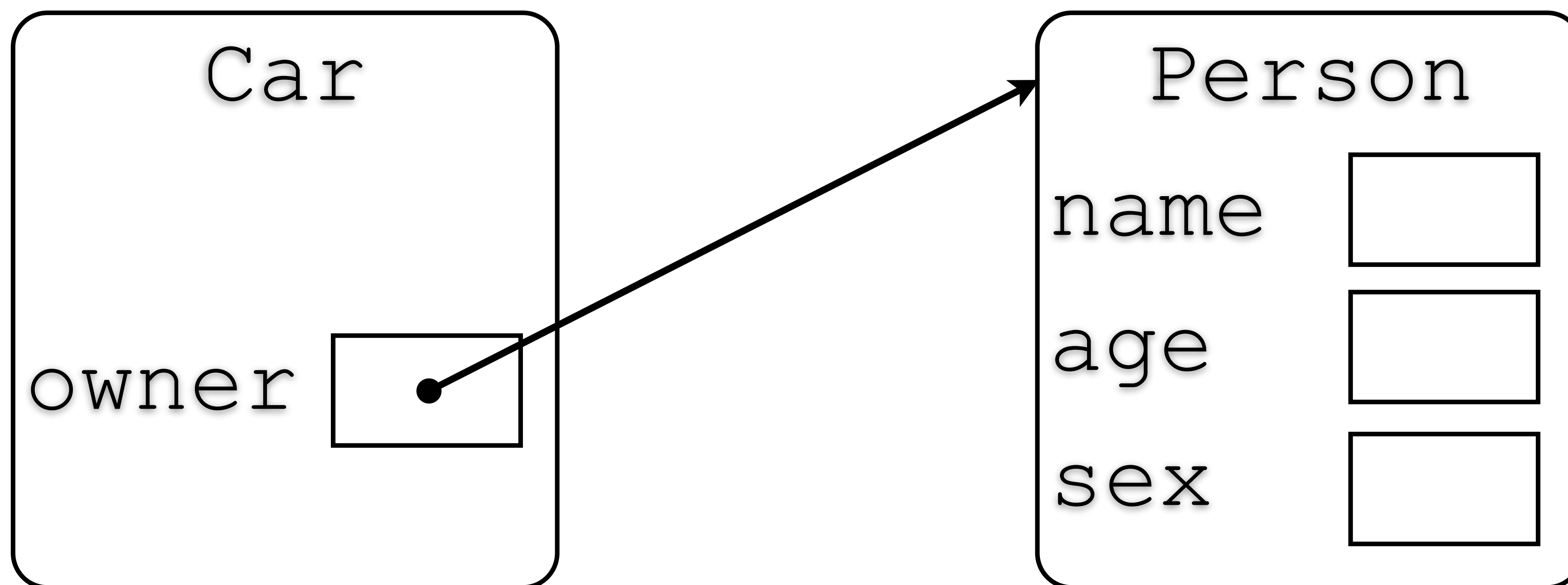
Composition: objects combining to create more complex ones

Inheritance: objects are created as extensions of other objects with additional properties

Association

In an associative **has-a** relationship, an object is aware of another complex object and can communicate with it.

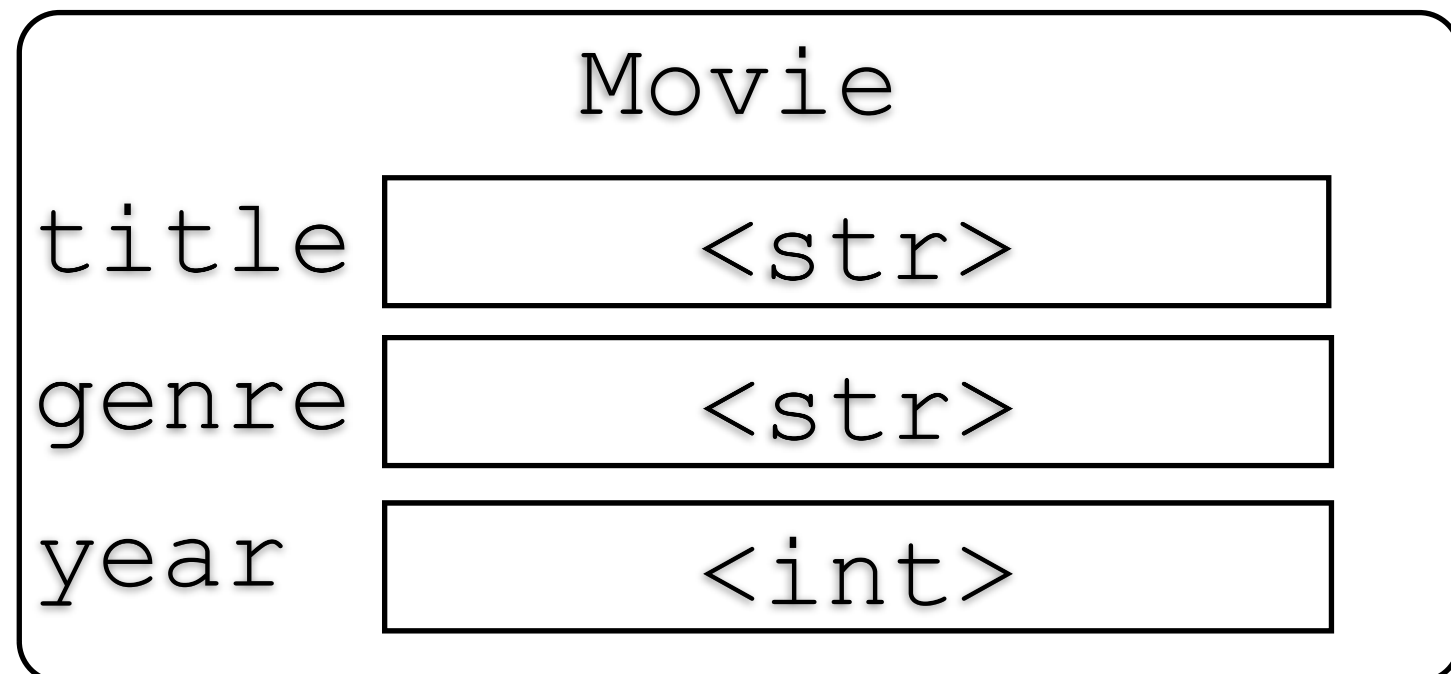
Example: a `Car` has an `owner` attribute which is a `Person`.



Composition

In a compositional **has-a** relationship, an object is made up of less complex objects.

Example: a `Movie` object is composed of string objects `title` and `genre` and integer object `year`.



Inheritance

Inheritance, as opposed to the previous two examples, is not a has-a relationship. It's an **is-a** relationship.

It means that objects of a particular class are members of a subset of the objects of another class.

The subclass inherits all the properties of the superclass, and adds some of its own.

Inheritance is a largely misunderstood idea, so we will spend a bit of time clarifying when it is useful and when it is not.

Inheritance Example

Consider the class `Person`:

```
class Person():
    def __init__(self,n, y, g):
        self.name = n
        self.year = y
        self.gender = g
```

Several modules may use this class to keep track of `Person` objects.

Now, imagine that the university would like to use the `Person` class to store information about its students.

Inheritance Example

The `Person` class does not have all the attributes necessary to keep track of a student's personal information.

What can we do?

We could add what we need to `Person`, which, if done by all other methods that may be using `Person`s, would make `Person` a very long, unwieldy class.

Alternatively, we can create the parts that we're missing (student number, GPA, etc.) in another class and connect it to the `Person` class somehow.

The `Student` Class

We create the class `Student`:

```
class Student():
    def __init__(self, stn, avg):
        self.student_number = stn
        self.gpa = avg
```

Now, this `Student` class also needs a name, a gender and a year of birth.

We have three options:

The `Student` Class - Option A

```
class Student():
    def __init__(self,n,y,g,s,a):
        self.name = n
        self.year = y
        self.gender = g
        self.student_number = s
        self.gpa = a
```

This option makes all the `Person` functionality available in the `Student` class, but it has a drawback: if a new attribute needed to be added for all people, it would have to be added in two places.

The `Student` Class - Option B

```
class Student():
    def __init__(self,n,a,p):
        self.student_number = n
        self.gpa = a
        self.person = p
```

This option makes the `Student` class store a `Person` class in `self.person`. This way, looking for the `Student`'s name would involve checking its `person` attribute's name.

However, this is counter-intuitive as a metaphor since the student is not a separate entity from the person. It's not that students HAVE people, students ARE people.

The `Student` Class - Option C

What if there was a way to express that every Student was a Person with extra information, that students were a subset of people?

We specify that `Student` inherits from `Person` by giving person as a parameter to the class definition:

```
class Student(Person):
```

This means that the `Student` class automatically takes on all the properties of the `Person` class before any of its own are even defined.

The `Student` Class

Then, we add what we're missing and pass on pertinent information to our parent/ancestor/ superclass:

```
class Student(Person):
    def __init__(self,n,y,g,sn,a):
        Person.__init__(self,n,y,g)
        self.student_number = sn
        self.gpa = a
```

The bold line initializes the `Person`-specific parts of our `Student`. It uses standard method syntax.

Inheriting Attributes

Let's make a new `Student`:

```
>>> ramona = Student("Ramona",\
1987, 'F', 990000001, 3.0)
```

Our student has student-specific attributes:

```
>>> ramona.gpa
4.0
```

However, she also has all the attributes a person may have:

```
>>> ramona.name
Ramona
```

Inheriting Methods

All the `Person`'s methods are now `Student` methods as well, so if `Person` had a `__str__`:

```
def __str__(self):
    return "%s (%s) b. %s" %\
(self.name, self.gender, self.year)
```

Even though we haven't specifically defined a `__str__` method in `Student`, we have one:

```
>>> print ramona
Ramona (F) b. 1987
```

Overriding Methods

It's natural that a `Student`'s string representation would be different from a `Person`'s. So, if we wrote a `__str__` method for `Student`:

```
def __str__(self):
    return "Student %s (%s)" %\
(self.name, self.student_number)
```

This method would **override** (be called instead of) any method of the same name for its ancestor:

```
>>> print ramona
Student Ramona (990000001)
```

Overriding Methods

When overriding, we can still rely on the parent's method to do part of the work:

```
def __str__(self):
    return "Student " + \
                  Person.__str__(self)
```

Then, the `Person __str__` method would help build the result of the `Student __str__` method.

```
>>> print ramona
Student Ramona (F) b. 1987
```

Extending Functionality

We will want to do things with `Student`s that don't apply to all `Person`s. So, by writing methods in the `Student` class itself, we can extend functionality without affecting `Person`:

```
def raise_gpa(self, bonus):
    self.gpa += bonus

>>> ramona.raise_gpa(0.5)
>>> ramona.gpa
3.5
>>> velian = Person("Velian",
                                   1986,'M')
>>> velian.raise_gpa(0.5) <-- ERROR
```

Inheritance: Conclusion

Inheritance is one of the most powerful concepts in object-oriented programming, but it's also one of the most abused.

When we say that class B inherits from class A, we are making a very specific claim about the relationships between these two objects:

We are claiming that the objects in class B are a subset of the objects of class A, and have all their properties and more.

Cars are objects. People own cars. Why don't we let `Person` inherit from `Car` to represent people who own cars?........

I wouldn't.