Inheritance

Updated Sept 18, 2pm

Consider the following problem description, which is adapted from an old midterm question.

- A building has an address, and a number of rooms, provided at the time of construction.
- A room has a name (an arbitrary string) and a size, provided at the time of construction.
- When printed, a building prints the sum of the square footages of all of its rooms.
- You can add a new room to a building.

A class design for this problem might look like the following:

```
class Building:
    def __init__(self, address, rooms):
        """ (Building, str, list of Room) -> NoneType """
        self.address = address
        self.rooms = rooms

    def __str__(self):
        """ (Building) -> str """
        sum = 0
        for room in self.rooms:
            sum += room.size
        return str(sum)

    def add_room(self, room):
        """ (Building, Room) -> NoneType """
        self.rooms.append(room)

class Room:
    def __init__(self, name, size):
        """ (Room, str, float) -> NoneType """
        self.name = name
        self.size = size
```

Not too bad. But now, consider the following extension:

- A **house is a type of building** with at most 10 rooms, and prints "Welcome to our house", plus the details of all of its rooms (name and square footage, separated by commas).

Note that we don't want to create an *object* called "house" of class `Building`, because we don't want just one house! What we really want is to create a new `House` class. But we don't want to start from scratch, because we'd end up repeating a lot of code that's already in the `Building` class. Instead, we want to base the `House` class on `Building`, because every House **is a** Building.

In object-oriented programming, we can do this using **inheritance**, which allows us to

create new classes by specialising existing ones. Let's see how to do this in Python:

```
class House(Building):
    def __init__(self, address, rooms):
        """ (House, str, list of Room) -> NoneType """
        if len(rooms) > 10:
            raise TooManyRoomsError
        else:
            self.address = address
            self.rooms = rooms

    def __str__(self):
        s = 'Welcome to our house\n'
        for room in self.rooms:
            s += '{}, {}\n'.format(room.name, room.size)
        return s
```

There's quite a bit of new syntax in this code, so let's break this down.

- The first line `class House(Building)` creates a new class that is a **subclass** of `Building`. We can also say that `Building` is a **superclass** of `House`.

- This means that every method of `Building` is also a method of `House`! (We say that `House` objects "inherit" the methods of class `Building`.) For example, the following code works just fine, because `House` objects still have access to the `add_room` method defined in `Building`:

  ```
  >>> my_house = House('101 My Street', [Room('Living Room', 100)])
  >>> my_house.add_room(Room('Bedroom', 75))
  ```

- You might notice that the `__str__` method now gets defined twice for the `House` class: once in the body of `Building` and once in the body of `House`. This is an instance of an **overridden** method: the definition in the `House` class *overrides* the one from `Building`, and is executed when `__str__` is called.

  ```
  >>> str(my_house)
  'Welcome to our house
  Living Room 100
  Bedroom 75'
  ```

Calling superclass methods

One issue that comes up with overriding methods is that the subclass seems to have "lost" access to the method of its superclass. What if, in the above example, we wanted to access the `Building` `str` method from a `House` object? Just calling `str(my_house)` wouldn't work, as this would call the overridden version. However, we can accomplish this using the following explicit method call (notice that the house object is now passed as the explicit `self` parameter):

```
>>> Building.__str__(my_house)
'175'
```

We can do this inside a class definition, too. Let's rewrite the `House` constructor to avoid the redundant code in the `else` block:

```
class House(Building):
    def __init__(self, address, rooms):
    if len(rooms) > 10:
        raise TooManyRoomsError
    else:
        Building.__init__(self, address, rooms)
```

When to use inheritance

Inheritance is a powerful tool in designing your code. In fact, Python is built from inheritance. Every built-in Python object is a descendant of the base `object` class, which provides methods like `__str__` and `__init__` (and all others with the double underscore).

By using inheritance, you can eliminate redundant code and set up templates that you can use to organize your code. If you take CSC207, you'll discuss object-oriented design in much greater detail, and probably use inheritance on a much larger scale than we're doing here.

However, inheritance isn't perfect for every situation. An alternative to inheritance is **composition**, in which classes simply store references to objects of other classes if they want to make use of their functionality. This is commonly thought of as a "has a" relationship rather than the "is a" relationship of inheritance. For example, to represent people who are car owners, a `Person` object might have an attribute `car` which stores a reference to a `Car` object. We probably wouldn't use inheritance to store the relationship between `Person` and `Car`!

Of course, the "has a" vs. "is a" categorization is rather simplistic, and not every real-world problem is so clearly defined. One more technical issue with inheritance is that it binds classes tightly together: any change in a superclass affects all of its subclasses, which can lead to unintended effects. In your next lab, you'll touch on multiple inheritance, a powerful Python feature that many avoid because of its subtleties.

Quick comments from lecture

1. A few students asked questions related to the "flow of information": can code in a subclass affect the superclass? The answer is **no**; if we create an instance of a `Building` class, no code in `House`, whether it be a new method or an overridden method, will affect that object.

2. We briefly saw that we can create hierarchies of inheritance:

```
class Townhouse(House):
    def __init__(self, address, rooms):
        House.__init__(self, address, rooms)

>>> t = Townhouse(House)
>>> str(t) # Exercise: try this out
```

Exercise

Extend the code from this page to add the following functionality:

- An office building may have any number of rooms, but no room may be named

`'Bedroom'`, or have a square footage of less than 100.

- It is possible to rename and room in any building (by specifying an old and a new name), but only an office building can change the square footage of their rooms (by specifying the room name and the new square footage). Write a main code body (that should only execute when this file is run directly, not when it is imported), to perform the following:
- prompt the user for a type of building, address, and number of rooms
- prompt the user for room names and square footages until all rooms have been named

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)