Lezione 5: Programmazione orientata agli oggetti, ereditariet e polimorfismo.

Rispondere in maniera sintetica alle seguenti domande ed esercizi. Le risposte possono essere discusse in orario di ricevimento, da concordare con il docente. Le domande/esercizi sono formulate in inglese per facilitare il riuso del materiale didattico.

1 Questions

Question 1. Why might a class need to manually call the \_init\_ method in a superclass?

Question 2. How can you augment, instead of completely replacing, an inherited method?

Question 3. Why is it better to customize by subclassing rather than copying the original and modifying?

Question 4. Why is it better to call back to a superclass method to run default actions, instead of copying and modifying its code in a subclass?

2 Exercises

Exercise 1. Write a class called Adder that exports a method add(self, x, y) that prints a Not Implemented message. Then, define two subclasses of Adder that implement the add method:
- **ListAdder**: with an `add` method that returns the concatenation of its two list arguments;

- **DictAdder**: with an `add` method that returns a new dictionary containing the items in both its two dictionary arguments (any definition of addition will do).

Experiment by making instances of all three of your classes interactively and calling their `add` methods. Now, extend your `Adder` superclass to save an object in the instance with a constructor (e.g., assign `self.data` a list or a dictionary), and overload the `+` operator with an `__add__` method to automatically dispatch to your `add` methods (e.g., `X + Y` triggers `X.add(X.data, Y)`). Where is the best place to put the constructors and operator overloading methods (i.e., in which classes)? What sorts of objects can you add to your class instances?

In practice, you might find it easier to code your `add` methods to accept just one real argument (e.g., `add(self, y)`), and add that one argument to the instance's current data (e.g., `self.data + y`). Does this make more sense than passing two arguments to `add`? Would you say this makes your classes more object-oriented?

**Exercise 2.** Make a subclass of `Mylist` from exercise 3 homework 4 called `MylistSub`, which extends `Mylist` to print a message to `stdout` before each overloaded operation is called and counts the number of calls. `MylistSub` should inherit basic method behavior from `Mylist`. Adding a sequence to a `MylistSub` should print a message, increment the counter for `+` calls, and perform the superclass method. Also, introduce a new method that prints the operation counters to `stdout`, and experiment with your class interactively. Do your counters count calls per instance, or per class (for all instances of the class)? How would you program the other option?

**Exercise 3.** Code a set of six class statements to model the taxonomy shown in Fig. 1 with Python inheritance. Then, add a `speak` method to each of your classes that prints a unique message, and a reply method in your top-level `Animal` superclass that simply calls `self.speak` to invoke the category-specific message printer in a subclass below (this will kick off an independent inheritance search from `self`). Finally, remove the `speak` method from your `Hacker` class so that it picks up the default above it. When you're finished, your classes should work this way:

```python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
```
>>> spot.reply()
meow

>>> data = Hacker()

>>> data.reply()
Hello world!

![Diagram of class hierarchy](image)

**Figura 1:** Hierarchy of classes in the animal domain.

**Exercise 4.** Consider the object embedding structure captured in the figure Fig. 2. Code a set of Python classes to implement this structure with composition. Code your **Scene** object to define an **action** method, and embed instances of **Customer**, **Clerk**, and **Parrot** classes all three of which should define a **line** method which prints a unique message. The embedded objects may either inherit from a common superclass that defines line and

![Diagram of scene composite](image)

**Figura 2:** A scene composite.
simply provide message text, or define line themselves. In the end, your classes should operate like this:

```python
>>> import parrot
>>> parrot.Scene().action()
customer: "that's one ex-bird!"
clerk: "no it isn't..."
parrot: None
```