Inheritance

csc148, Introduction to Computer Science
Diane Horton
Winter 2015

UNIVERSITY OF
TORONTO

DCS50

Example

- Context: A company payroll system.
- There are several kinds of employees:
  - those paid hourly
  - those paid a salary
  - those paid on commission
- We need to be able to do things like:
  - compute how much a person should be paid in a given pay period
  - keep track of a person's pay history

Option 1: Three different classes

- We could write a class for each kind of employee.
- Each class would bundle together the data and methods we need for that kind of employee.
- Data:
- Methods:
- Do you like this design?

Option 2: They share a "parent" class

- We can say *in our code* that all three are kinds of employee.
- How we do that:
  - Define a class called `Employee`. It is the parent class.
  - in each of the three original classes, we say it is a child class of Employee. Example:
    `class SalariedEmployee(Employee)`
- A `SalariedEmployee` inherits all the data and methods of `Employee`.

We customize the child classes

- A child class can, for example:
  - Extend the parent, by adding attributes and/or methods.
  - Override the parent, by re-defining an existing method.
- It can still call methods in the parent: *«Parent»*.*«method»*(self, …)
- [The code for our employee classes…]
- Do you like this design more or less than the first one, with no common parent class?

A canonical use of inheritance

- The parent class can’t define the body of all methods.
  - It should therefore not be instantiated!
  - The incomplete methods raise an exception to warn client code it is doing something wrong.
- Child classes define those incomplete methods.
- The parent class is still useful.
  - It defines what all children must do.
- Important: We can call methods on an `Employee` without knowing which kind it is!

Things to notice about `Company.py`

- It calls `record_pay` and `total_pay` on objects without knowing what kind of `Employee` they are.
  - Every kind of `Employee` has those methods.
- But it only calls `log_hours` on an object that is specifically a `HourlyEmployee`.
  - Other kinds of `Employee` don’t have that method.
- And it never constructs a plain old `Employee`.
  - That class is “abstract”: it has methods that will raise an exception if called!

Things to notice about `Employee.py`

- It can't implement `amount_of_pay`
  - It depends on knowing details only available in a child class.
- But it can implement `record_pay` (with the help of `amount_of_pay`).
  - That's why I separated these two methods.
- Most of the instance variables are public, so are in the class docstring. But not `pay_history`.
  - Info about it is provided by a method instead.
  - This is a design decision.

Notice about `HourlyEmployee.py`

- It inherits all methods from `Employee`, but ...
- It overrides `__init__`.
  - It calls its parent's `__init__`, then adds on.
- It overrides `amount_of_pay`.
  - It finally can give a meaningful implementation.
- It overrides `record_pay`.
  - It calls its parent's `record_pay`, then adds on.
- It extends its parent by adding `log_hours`.

Inheritance & Finding methods/attributes

- When we say `thingee.stuff` or `thingee.do_something()`, Python must:
  1. Find the name `thingee`.
  2. Follow the reference in it, to get to an object.
  3. Look inside the object to find attribute `stuff` or method `do_something`.
- Suppose `thingee` is both a `PencilCase` and a `Container`, because of inheritance.
  - There may be more than one definition of `stuff` and `do_something`!

How Python does it

- Python starts looking in the most specific part of the object.
  - If not found, it goes "up" as needed.
- Suppose a method in a parent class calls a helper method.
  - Python still starts looking in the most specific part of the object.
- Example: next slide.
- Trace it in the visualizer.

```
class A:
   def g(self, n):
      return n
   def f(self, n):
      return self.g(n)

class B(A):
   def g(self, n):
      return 2 * n


a = A()
b = B()
print("a.f(1): {}".format(a.f(6)))
print("b.f(1): {}".format(b.f(6)))
```