Inheritance

CSC148, Introduction to Computer Science
Diane Horton
Fall 2017
Example

• Context: A company payroll system.
• There are two kinds of employees:
  • those paid hourly
  • those paid a salary
• We could define two unrelated classes.
• But we’d like to do things for all employees, e.g.,:
  for employee in employees:
    employee.pay(date(2017, 9, 30))
Inheritance to the rescue!

- Create a general class called `Employee`.
  - It defines the methods and attribute that all types of employee have in common.
  - This is called the **shared public interface**.
- Define a more specific subclass of `Employee` called `SalariedEmployee`.
  - It will **inherit** all methods and attributes of `Employee`.
  - It can modify or add to these as needed.
- Do the same for `Hourly Employee`. 
We’ll start with methods

• We said that subclasses will inherit methods and attributes.
• In fact, inheriting attributes is not automatic in Python. We have to make it happen.
• So we’ll write our code first with just methods, and add attributes afterwards.
• (This will mean we hard-code some values at first.)
… we developed code …
Why bother with the abstract class?

• Client code should not make an instance of Employee.
  • The docstring warns against it.
  • Reason:
• But class Employee is still useful!
  • Client code with some kind of Employee object knows it can call pay or get_monthly_payment.
  • A programmer defining a future subclass of Employee knows it must override method get_monthly_payment (unless the new class is also to be abstract).
Four things to do with an inherited method

- **Inherit** the method as is.
- **Override** the method in one of these ways:
  - Complete an unimplemented method.
  - Replace the body of the inherited method.
  - Extend the behaviour of the inherited method. (I.e., replace its body, but include a call to the parent’s method as a helper.)
Which one?

• Suppose class `Monster` has an `eat` method, and its child class `Dragon` overrides that method.

• If we say `fluffy.eat()`, which `eat` method is called?

• The one defined in the class that `fluffy` was constructed from.

```
fluffy = Monster()  fluffy = Dragon()
fluffy.eat()         fluffy.eat()
```

• If there is none there, we look in the parent, and if there is none there we look in its parent and so on.
Initializers

• **Employee** warns not to call **__init__**.
• This warning is for client code.
• Child classes *should*, because it initializes attributes that all instances must have.
  • The alternative is to duplicate code. Yuk.
• Note the syntax for doing so. In a child class such as **HourlyEmployee** we write:
  ```python
  Employee.__init__(self, . . .)
  ```
Methods are inherited, attributes are not

- An instance of a class inherits all the methods from its parent class.
- It can accept each one as is, or override it.
- An instance *does not* inherit the attributes of its parent class.
- Attributes come into being when we initialize them.
- We can get all the attributes of the parent class by calling its initializer.
Example

• An instance of a `SalariedEmployee` inherits methods `pay` and `get_monthly_payment` from `Employee`.
  
  • It can accepts `pay`.
  
  • It overrides `get_monthly_payment`.

• It does not inherit the attributes `id_`, `name`, and `pay_history` of its parent `Employee`,

• It gets these by calling `Employee.__init__`. 
Special methods

- Constructing an object automatically calls `__init__`
- Similarly, `print(x)` automatically calls `x.__str__()`
- Every class inherits an `__init__` and `__str__` from class `object`. (Not a very useful one!)
- We can override it with something specific to our class.
- These underscored methods are called special methods.
- We’ll see more of these later.
Why do these work?

for e in employees:
    e.pay(date(2017, 9, 30))

def lifetime_pay(e: Employee) -> float:
    total = 0
    for amount in e.pay_history.values():
        total += amount
    return total

Each e could be a salariedEmployee or an hourlyEmployee or something as yet undefined!
Polymorphism

• … Because Employee defines a shared public interface for itself and its descendants.

• All instances have that shared interface, even though they can take different forms.

• We call this polymorphism (“many forms”).
When to use inheritance

• We will focus on using inheritance to define a common shared interface.
• There are other uses you’ll learn later, starting in csc207.
• Another way to describe an inheritance relationship: \texttt{HourlyEmployee is-a Employee}.
• But inheritance isn’t everything. Don’t forget has-a!