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:
    ```python
class SalariedEmployee(Employee)
```
  • A SalariedEmployee inherits all the methods of Employee.
We customize the “child” classes

• A child class **inherits** all methods of its parent.
• A child class can, for example:
  • **Extend** the parent, by adding attributes and/or methods.
  • **Override** the parent, by re-defining an existing method.
An abstract class

• 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.
• Any class with at least one non-implemented method is an abstract class.
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!
Polymorphism

• This works:
  ```python
  for e in weekly_employees:
    print(e.amount_of_pay())
  ```
because Employee defines a shared public interface for itself and its descendants.

• All of the e’s above have a common interface, even though they can take different forms
  • as a salariedEmployee or an hourlyEmployee or something as yet undefined!

• We call this **polymorphism** (“many forms”).
Things to notice about `Employee.py`

- It can’t implement `amount_of_pay`
  - This 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, 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`. 
Things to notice about the initializers

• **Employee** does implement `__init__`, but carries a warning not to call it.
• But child classes are not client code, so they may call it.
• And they *must*, because it initializes attributes that all instances must have.
• Note the syntax for doing so. In a child class such as `HourlyEmployee` we write:
  ```python
  Employee.__init__(self, ...)
  ```
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: HourlyEmployee is-a Employee.
• But inheritance isn’t everything. Don’t forget has-a!
Method/attribute look-up

• 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)))