January 20, 2025

LEC 03 - Inheritance

- Suppose we have a `SalariedEmployee` class and we want to make a new kind of employee: `HourlyEmployee`.
- The specs for HourlyEmployee would be very similar
    - Same attributes: id_, name
    - Same methods: `get_monthly_payment()`, `pay()`
    - Slight differences: salary vs. hourly wage + hours worked
- We could try copy, pasting, and modifying `SalariedEmployee` ⇒ `HourlyEmployee`
    - Introduces a lot of duplicate code, which makes it hard to update the classes and fix bugs
- We could try using composition by including a `SalariedEmployee` object in the `HourlyEmployee` class and reusing the attributes and methods
    - Overcomplicates the class since they require two different types of methods and attributes regarding payment
- It would be ideal to have an `Employee` class with common features to both `HourlyEmployee` and `SalariedEmployee`
    - Also allows even more kinds of employees to easily be added

---

Inheritance:

- Factor out common features and write them only once in a base / abstract / parent / super class
- `SalariedEmployee` and `HourlyEmployee` are subclasses of `Employee`
-

Abstract Classes:

Interfaces:

- An abstract class is first and foremost the explicit representation of an interface in a Python program

Shared Implementations:

- An abstract class also enables the sharing of code through method inheritance
- Most methods will be left up to the subclass to implement in that context

```
raise NotImplementedError
```

- Some methods can be implemented in the abstract class, if behaviour will be identical in the subclasses anyways

January 20, 2025

LEC 03 - Inheritance

Class Design With Inheritance:

Ask yourself:

- What attributes and methods should comprise the shared public interface
- For each method, should its implementation be shared or separate for each subclass

The Four Cases of Method Inheritance:

- Subclasses use several approaches to recycling code from the superclass, using the same name
    1. Subclass ***inherits*** superclass method
    2. Subclass ***overrides*** an abstract method (to ***implement*** it)
    3. Subclass ***overrides*** an implemented method (to ***extend*** it)
    4. Subclass ***overrides*** an implemented method (to ***replace*** it)

Polymorphism:

- Suppose a company has a list of employees
    - Some could be salaried, others hourly
- One code to rule them all
    - Same code to pay an employee regardles of their type

```
class Company:
      """..."""
      employees: list[Employee]

      def pay_all(self) -> None:
            for emp in self.employees:
                  emp.pay(date.today())
```

- Polymorphism allows employees of all types to be treated as instances of employee
    - They can be stored in a `list[Employee]` AND a `list[SalariedEmployee/Employee]` respectively
    - They can be iterated over with a single loop