Inheritance

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 9B's technical content. See python-inheritance[2] for the full reference on subclassing, `super()`, and polymorphism.

Today's outline

- Inheritance and "is-a" relationships
- Terminology: subclass/superclass, child/parent, extends
- Inheriting, introducing, and overriding methods
- `super()` and extending methods
- Polymorphism

Inheritance

Inheritance is another way of reusing code in classes. By inheriting from a class, we can create specialized (sub)classes while reusing attributes/methods from the original class. Key idea: building a hierarchy of classes.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-inheritance.pdf

```
        Robot
   /      |       \
DeliveryRobot  CleanerRobot  HumanoidRobot
```

This is an **"is-a"** relationship: a `DeliveryRobot` *is a* `Robot`. `DeliveryRobot`, `CleanerRobot`, and `HumanoidRobot` all specialize the base class (`Robot`):

- They inherit the attributes and methods of the base class.
- They can **override** the attributes/methods of the base class (e.g. a `move()` method might have different implementations).
- They can have new (unique) attributes/methods or implementations.

More examples of "is-a" hierarchies:

- `Employee` → `Programmer`, `Manager`, `Admin`
- `Animal` → `Cat`, `Dog`, `Fox`; and `Dog` → `ServiceDog`
- `Person` → `Professor`, `Student`; and `Student` → `UndergradStudent`, `GradStudent`

Terminology

Given `class Student(Person):` — all of the following mean the same thing, and can be used interchangeably:

- The `Student` class **inherits** from the `Person` class.
- The `Student` class is a **subclass** of the `Person` class.
- The `Student` class is a **child class** of the `Person` class.
- The `Person` class is a **superclass** of the `Student` class.
- The `Person` class is a **parent class** of the `Student` class.
- (less often) The `Student` class **extends** the `Person` class.

Reusing code via inheritance

Without inheritance, two robot classes end up duplicating `__init__` and `charge`:

```
# BAD -- duplicating code, violating the DRY principle
class CleaningRobot:
    def __init__(self, name: str, batt_level: int):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
```

```
        print(f"{self.name} is fully charged!")

    def clean(self):
        print(f"{self.name} is cleaning the floor.")

class DeliveryRobot:
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        self.name = name
        self.batt_level = batt_level
        self.load_capacity = load_capacity

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")

    def deliver(self):
        print(f"{self.name} is delivering a package up to {self.load_capacity}kg.")
```

Instead, extract the common behaviour into a **base class**, and inherit specialized robots from it:

```
class Robot:
    def __init__(self, name: str, batt_level: int):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")


class CleaningRobot(Robot):
    def clean(self):
        print(f"{self.name} is cleaning the floor.")
```

The syntax `class Child(Parent):` means the child class inherits from the parent class.

```
>>> cleaner = CleaningRobot("Roomba", 80)   # calls Robot's (base class') __init__
>>> cleaner.charge()   # CleaningRobot inherited charge() from the base class
Roomba is fully charged!
>>> print(cleaner.name)   # CleaningRobot inherited attributes from the base class
Roomba
```

Introducing new methods

A child class can define methods that don't exist on the base class at all — `clean()` only exists on `CleaningRobot` objects, not on plain `Robot` objects.

Overriding methods

We can **override** a method by simply declaring one of the same name in the child class:

```
class FastChargingRobot(Robot):
    def charge(self):   # overriding the charge method from the base class
        self.batt_level = 100
        print(f"{self.name} is fully charged in record time!")
```

Calling `charge()` on a `FastChargingRobot` object calls *this* overridden method, not `Robot`'s.

Extending methods with `super()`

`super()` specifies that we want to use the method from the *superclass*, rather than the child class — useful for extending (rather than fully replacing) inherited behaviour:

```
class DeliveryRobot(Robot):
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        # super().__init__(...) calls the __init__ method from the parent (base) class
        super().__init__(name, batt_level)
        self.load_capacity = load_capacity

    def deliver(self):
        print(f"{self.name} is delivering a package up to {self.load_capacity}kg.")
```

```
>>> deliverer = DeliveryRobot("DHLBot", 80, 10)
>>> deliverer.charge()
DHLBot is fully charged!
>>> deliverer.deliver()
DHLBot is delivering a package up to 10kg.
```

Polymorphism

The word **polymorphism** means "to take on many forms". In computer science, an interface that works over different underlying data-types is called **polymorphic**.

`len()` is polymorphic — it works across many different types:

```
>>> len({'a': 1, 'b': 2, 'c': 3})
3
>>> len("Drop Bear")
9
>>> len([1, 2, 3, 4])
4
```

Classes that share a common interface (e.g. all inheriting a `charge()` method) are polymorphic too — the same call produces different behaviour depending on the actual object's class:

```
robots = [
    CleaningRobot("Roomba", 40),
    DeliveryRobot("DHL-Bot", 20, 15),
    FastChargingRobot("FlashBot", 5)
]

for r in robots:
    r.charge()   # different behaviors depending on r's charge() method
```

Exercise

Implement the `SurveyDrone` and `DeliveryDrone` classes, inheriting from `DroneFlight`:

```
>>> class DroneFlight:
...     pass   # implemented before (see [[2025-09-23-composition]])
>>> class SurveyDrone(DroneFlight):
...     pass   # implement SurveyDrone
>>> class DeliveryDrone(DroneFlight):
...     pass   # implement DeliveryDrone
```

Left unsolved in the source — just stubs.

Summary

We can create classes that **inherit** from other classes. By doing so, the subclass automatically receives all attributes and methods implemented in the superclass. The subclass can have additional methods and attributes, while overriding inherited methods and calling methods from its superclass.

Use inheritance only when there's a clear **"is-a"** relationship.

Next: Unified Modelling Language (Week 10).