CoPL
Object Oriented Programming

Crashcourse

Daan Wichmann

Radboud University

02 04 2025

Inheritance

New idea… new problem…

New idea... new problem...

- We have now modeled a person with our *Person* class

New idea... new problem...

- We have now modeled a person with our *Person* class
- What if we want to model a student?

New idea... new problem...

- We have now modeled a person with our *Person* class
- What if we want to model a student?
- A student is also a person!

New idea... new problem...

- We have now modeled a person with our *Person* class
- What if we want to model a student?
- A student is also a person!
  - **Duplicate code**

New idea... new problem...

- We have now modeled a person with our *Person* class
- What if we want to model a student?
- A student is also a person!
  - **Duplicate code**
- What is the best way to model this?

New idea... new problem...

- We have now modeled a person with our *Person* class
- What if we want to model a student?
- A student is also a person!
  - **Duplicate code**
- What is the best way to model this?
  - using **Inheritance!**

So, what is inheritance?

Inheritance

Lets look at the definition in the context of biology

Inheritance

Lets look at the definition in the context of biology

Inheritance (Biology)

Inheritance is the way that genetic information is passed from a parent to a child. Members of the same biological family tend to have similar characteristics

Inheritance

Lets translate this to OOP!

Inheritance

Lets translate this to OOP!

Inheritance (OOP)

Inheritance is the concept in which a class can *inherit* **data attributes** and **methods** from another class. Here, the class that inherits, is called the **derived class**. The class that the derived class inherits from, is called the **base class**.

Inheritance

Lets translate this to OOP!

Inheritance (OOP)

Inheritance is the concept in which a class can *inherit* **data attributes** and **methods** from another class. Here, the class that inherits, is called the **derived class**. The class that the derived class inherits from, is called the **base class**.

Alternative terminology

Inheriting is sometimes called **deriving**. So you can also say, the derived class **derives** data attributes and methods from it's base class.

Inheritance

In the case of our example:

Inheritance (OOP)

Inheritance is the concept in which a class (**Student**) can *inherit* data attributes (**first_name, last_name, age**) and methods (**speak()**) from another class (**Person**). The class that the derived class inherits from, is called the **base class**.

Alternative terminology

Inheriting is sometimes called **deriving**. So you can also say, the derived class **derives** data attributes and methods from it's base class.

What can we do with inheritance?

- Create a **hierarchy** of classes (i.e. a class hierarchy :D) which describes their relationship (e.g. a Student is a Person, but not the other way around)
- Avoid writing duplicate code!
- **Encapsulation** (more on that later)

Let’s look at the class hierarchy of our example!

Hierarchies

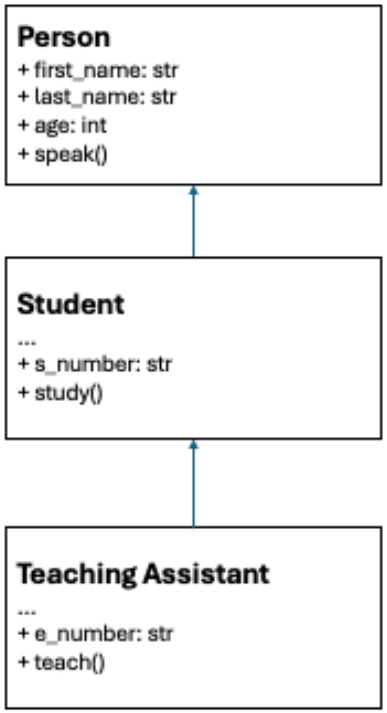

Figure: our hierarchy

Hierarchies

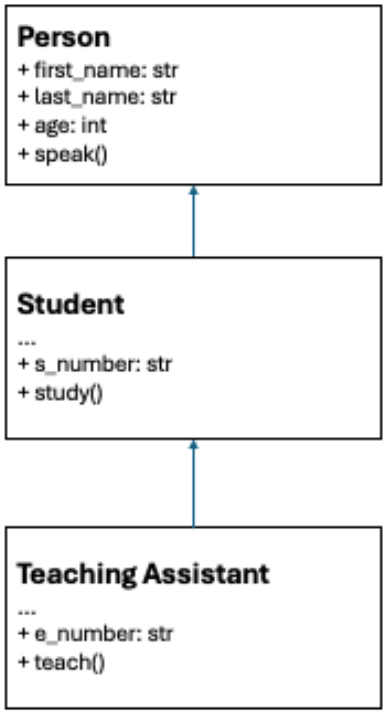

Figure: our hierarchy

Note: Inheriting classes (Student) can also be inherited from (TeachingAssistant)!

Surprise quiz (no grade)

Inheritance quiz

What is the *base class* of *Student*?

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

- **Answer:** TeachingAssistant

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

- **Answer:** TeachingAssistant

Which data attributes does *TeachingAssistant* inherit?

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

- **Answer:** TeachingAssistant

Which data attributes does *TeachingAssistant* inherit?

- **Answer:** first_name, last_name, age, and s_number

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

- **Answer:** TeachingAssistant

Which data attributes does *TeachingAssistant* inherit?

- **Answer:** first_name, last_name, age, and s_number

Which class is *TeachingAssistant* a *derived class* of?

Inheritance quiz

What is the *base class* of *Student*?

- **Answer:** Person

Which class is *Student* a *derived class* of?

- **Answer:** Person

What is a *derived class* of *Student*?

- **Answer:** TeachingAssistant

Which data attributes does *TeachingAssistant* inherit?

- **Answer:** first_name, last_name, age, and s_number

Which class is *TeachingAssistant* a *derived class* of?

- **Answer:** Person and Student (sorry, it's both)

How do we implement this in Python?

Remember our beautiful class

```python
class Person():

    # The constructor
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name  # Attribute
        self.last_name = last_name  # Attribute
        self.age = age  # Attribute

    # Method
    def speak(self):
        print(f"Hey, my name is {self.first_name}")
```

Let's derive from it

```
class Student(Person):
```

Let's derive from it

```python
class Student(Person):

    # The constructor
    def __init__(self, first_name, last_name, age, s_number):

        # Calling the constructor of base class
        super().__init__(first_name, last_name, age)
        self.s_number = s_number

    # Method
    def study(self):
        print(f"Student {self.s_number} is studying!")
```

The super() function

super()

The **super()** function refers to the base class of the class. We can use it to access/call methods of the base class.

If we call a method on the super() function, it is just like we would call it on **self**, but than as if it were an object of the base class!

Lets test it

Lets test it

```
student = Student("Daan", "Wichmann", "secret", "secret")
```

Lets test it

```
student = Student("Daan", "Wichmann", "secret", "secret")
student.speak()
>>> Hey, my name is Daan
```

Lets test it

```
student = Student("Daan", "Wichmann", "secret", "secret")
student.speak()
>>> Hey, my name is Daan
student.study()
>>> Student secret is studying!
```

Hierarchies - Another example

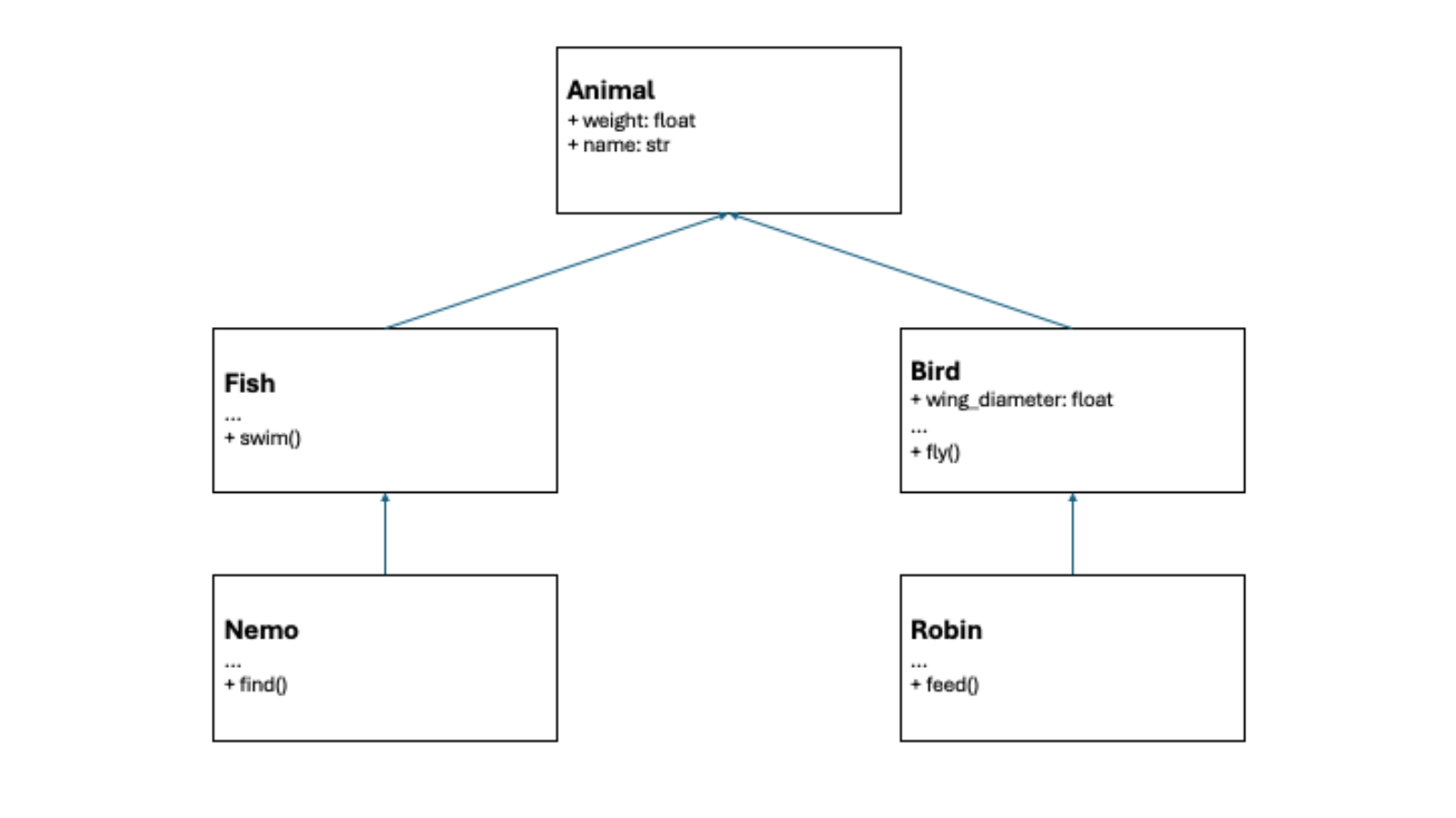

Figure: Another example

Method overriding

Overriding

- What if we want Student's *speak()* to do something different from Person's *speak()*?
- We can **override** the method! (**Method overriding**)

Our beautiful class again

```python
class Person():

    # The constructor
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name  # Attribute
        self.last_name = last_name  # Attribute
        self.age = age  # Attribute

    # Method
    def speak(self):
        print(f"Hey, my name is {self.first_name}")
```

And its subclass

```python
class Student(Person):

    # The constructor
    def __init__(self, first_name, last_name, age, s_number):

        # Calling the constructor of base class
        super().__init__(first_name, last_name, age)
        self.s_number = s_number

    # Method
    def study(self):
        print(f"Student {self.s_number} is studying!")
```

Currently

Currently

```
student = Student("Daan", "Wichmann", "secret", "secret")
```

Currently

```
student = Student("Daan", "Wichmann", "secret", "secret")
student.speak()
>>> Hey, my name is Daan
```

Overriding

- We want student to also mention their s-number when speaking!
- **Lets override speak()**

Overriding...

```python
class Student(Person):

    # The constructor
    def __init__(self, first_name, last_name, age,
    ↪  s_number):

        # Calling the constructor of base class
        super().__init__(first_name, last_name, age)
        self.s_number = s_number

    # Method
    def study(self):
        print(f"Student {self.s_number} is studying!")

    def speak(self):
        print(f"Hey, my name is {self.first_name} and my
        ↪  s-number is {self.s_number}")
```

And now

And now

```
student = Student("Daan", "Wichmann", "secret", "secret")
```

And now

```
student = Student("Daan", "Wichmann", "secret", "secret")
student.speak()
>>> Hey, my name is Daan and my s-number is secret
```

Operator overloading

Operator overloading

Operator overloading refers to a single operator's capacity to perform several operations based on the class (type) of operands

Operator overloading

Now in human terms:

Operator overloading

Now in human terms:

- Operators like addition (+), equals (==), and greater than (>) can perform certain actions based on their type/class.

Operator overloading

Now in human terms:

- Operators like addition (+), equals (==), and greater than (>) can perform certain actions based on their type/class.
- Other operator are:
  - +, -, /, //, %, ==, in, >, <, >=, <=, !=, and more!

Operator overloading

- Let's look at some examples

Operator overloading

- Let's look at some examples
- Addition:

Operator overloading

- Let's look at some examples
- Addition:
  - 5 + 3 = 8 (int)

Operator overloading

- Let's look at some examples
- Addition:
  - 5 + 3 = 8 (int)
  - "Cogn" + "AC" = "CognAC" (str)

Operator overloading

- Let's look at some examples
- Addition:
  - 5 + 3 = 8 (int)
  - "Cogn" + "AC" = "CognAC" (str)
- Greater than:

Operator overloading

- Let's look at some examples
- Addition:
  - 5 + 3 = 8 (int)
  - "Cogn" + "AC" = "CognAC" (str)
- Greater than:
  - 5 > 3 = True (int)

Operator overloading

- Let's look at some examples
- Addition:
  - 5 + 3 = 8 (int)
  - "Cogn" + "AC" = "CognAC" (str)
- Greater than:
  - 5 > 3 = True (int)
  - Person(...) > Person(...) = ???

Operator overloading

- We can define how these operators should behalve in our classes!

Operator overloading

- We can define how these operators should behalve in our classes!
- We use so called double underscore methods (dunder methods, you may forget that name).

What we want

- Consider two objects of our person class: person1 and person2.

What we want

- Consider two objects of our person class: person1 and person2.
- We want person1 > person2 = True, iff person1 is older than person2.

What we want

- Consider two objects of our person class: person1 and person2.
- We want person1 > person2 = True, iff person1 is older than person2.
- Let's **overload** the *greater than* operator in our class Person.

Let's overload

```
class Person():
    ...
```

Let's overload

```
class Person():
    ...
    def __gt__(self, other):  # Note the parameters
```

Let's overload

```
class Person():
    ...
    def __gt__(self, other):  # Note the parameters
        return self.age > other.age
```

Testing never ends

Let’s test again

Testing never ends

Let's test again

```
person1 = Person("Daan", "Wichmann", 19)
person2 = Person("Damai", "Jiworo", 18)
```

Testing never ends

Let's test again

```
person1 = Person("Daan", "Wichmann", 19)
person2 = Person("Damai", "Jiworo", 18)
print(person1 > person2)  # What will this return?
```

Testing never ends

Let's test again

```
person1 = Person("Daan", "Wichmann", 19)
person2 = Person("Damai", "Jiworo", 18)
print(person1 > person2)  # What will this return?
>>> True
```

Why is 'overloading'

Why is it called, overloading and not overriding?

Why is 'overloading'

Why is it called, overloading and not overriding?

- We are not overriding operators, because they should still behave the same for integers, strings, etc.

Why is 'overloading'

Why is it called, overloading and not overriding?

- We are not overriding operators, because they should still behave the same for integers, strings, etc.
- We are 'loading' the operator with functionality for more classes, i.e. **overloading operators**.

Part 2 of 3 done! Break?