Object Oriented Programming
Using
Python

Teemu Matilainen

teemu.matilainen@savonia.fi

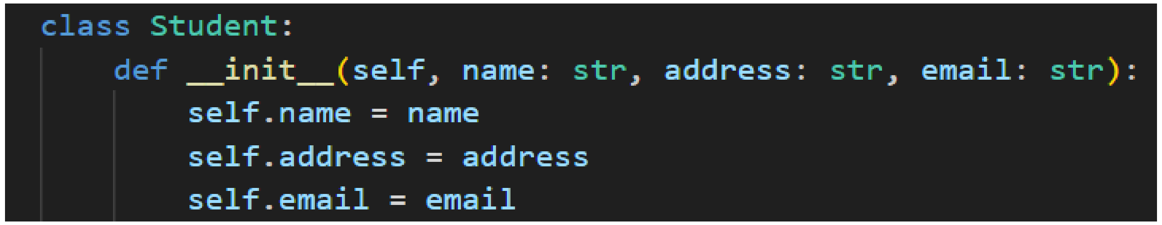
Lecture 6#

- You realize what inheritance is about
- You are able to inherit a class
- You are able to override a method in a subclass

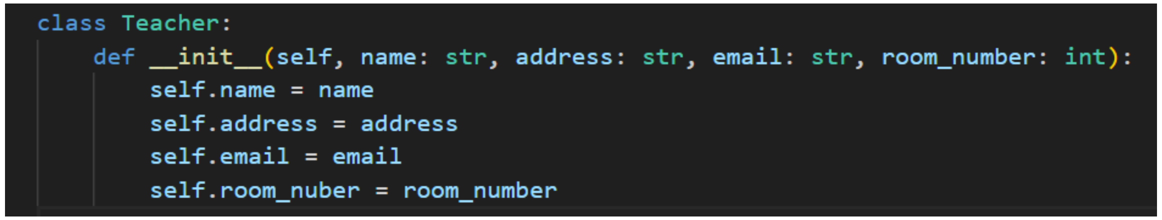
What inheritance is about?
How to inherit a class?

Tailoring Classes for Specific Needs...

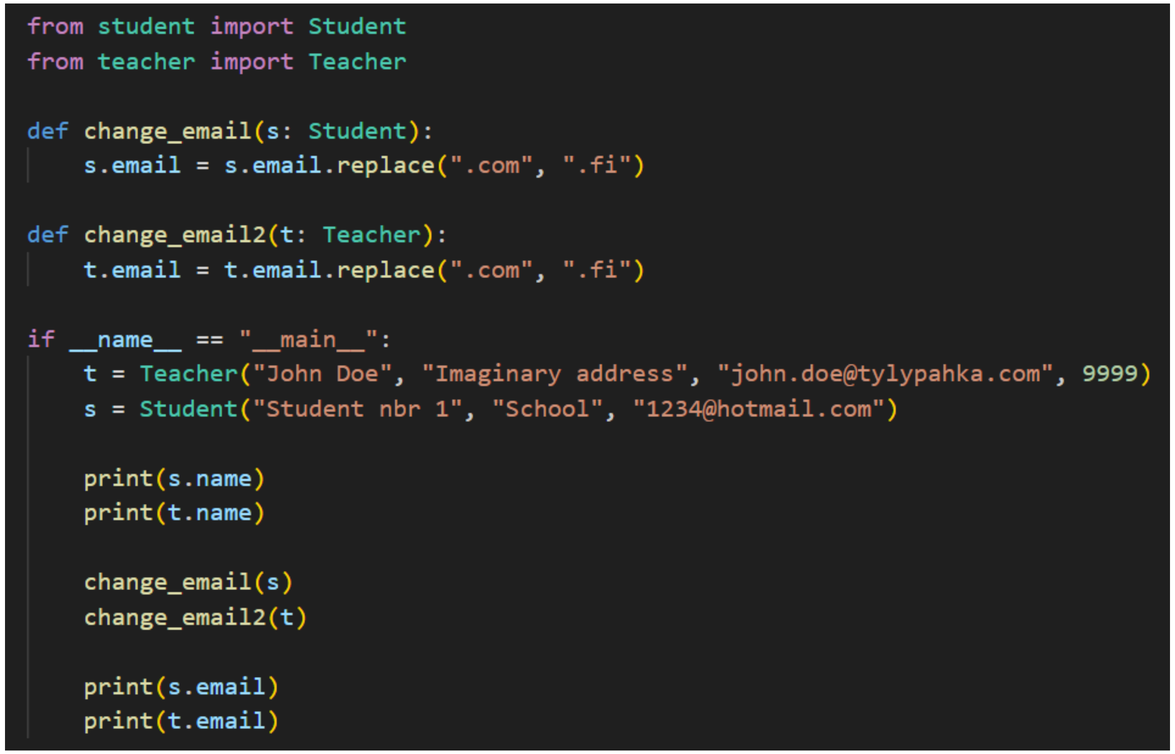
There are instances when you've already created a class, but later find the need for unique characteristics in certain instances, rather than all. Alternatively, you might notice that you've defined two closely related classes with only slight distinctions. As programmers, our goal is to minimize repetition while ensuring clarity and readability. How can we address varying implementations of inherently similar objects in such cases?



```python
class Student:
    def __init__(self, name: str, address: str, email: str):
        self.name = name
        self.address = address
        self.email = email
```



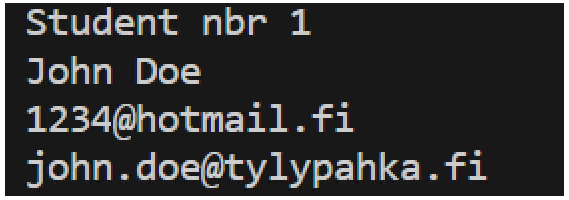
```python
class Teacher:
    def __init__(self, name: str, address: str, email: str, room_number: int):
        self.name = name
        self.address = address
        self.email = email
        self.room_nuber = room_number
```



```python
from student import Student
from teacher import Teacher

def change_email(s: Student):
    s.email = s.email.replace(".com", ".fi")

def change_email2(t: Teacher):
    t.email = t.email.replace(".com", ".fi")

if __name__ == "__main__":
    t = Teacher("John Doe", "Imaginary address", "john.doe@tylypahka.com", 9999)
    s = Student("Student nbr 1", "School", "1234@hotmail.com")

    print(s.name)
    print(t.name)

    change_email(s)
    change_email2(t)

    print(s.email)
    print(t.email)
```

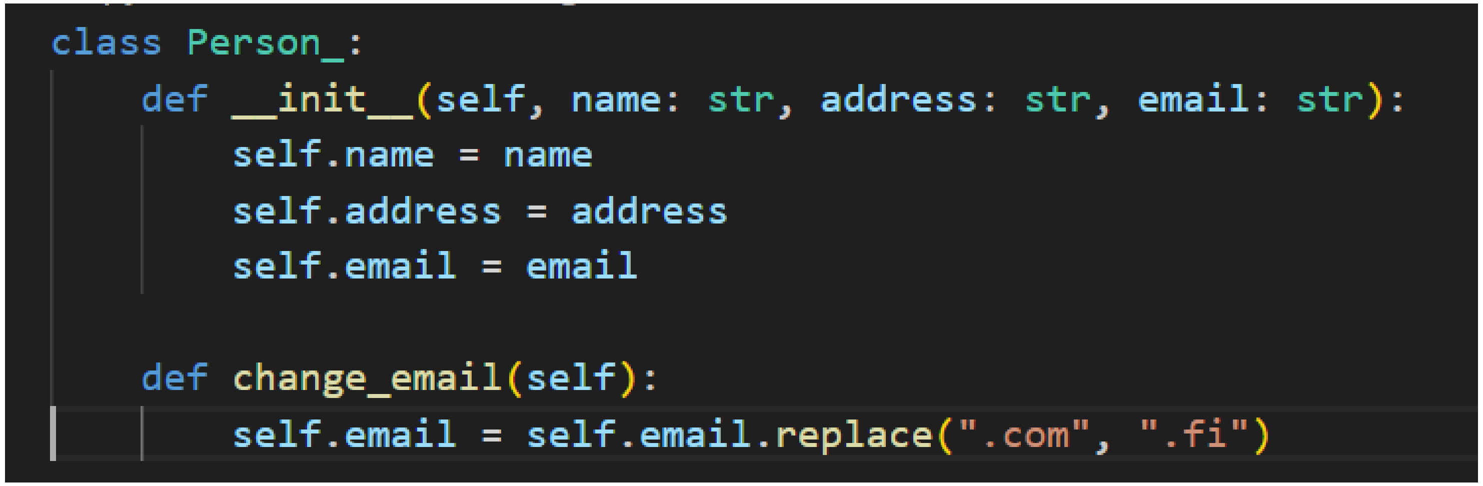
Example:

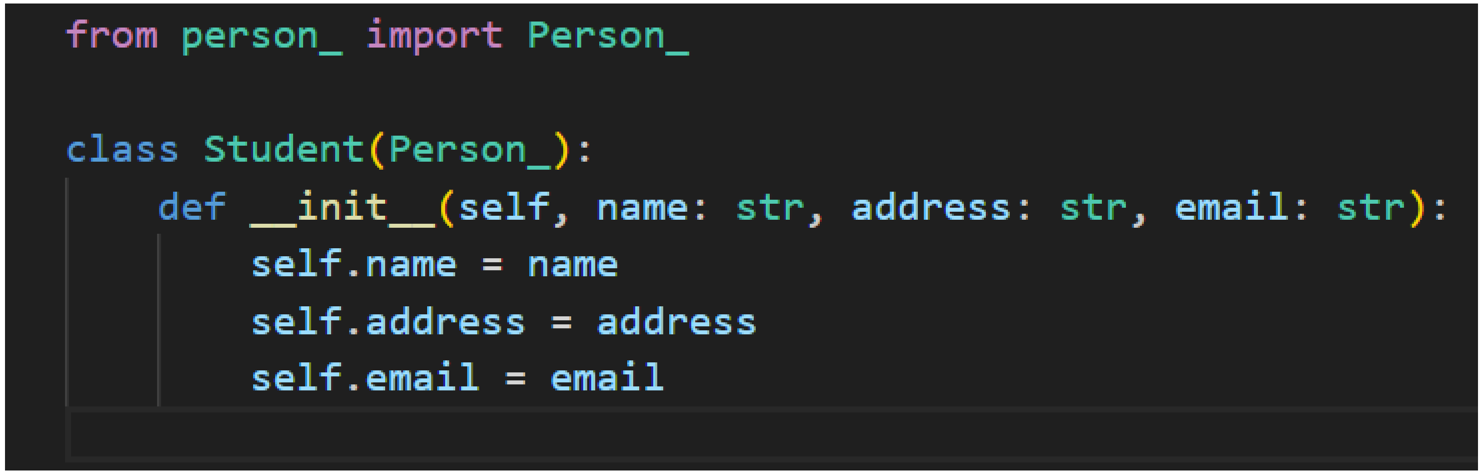
If we want to change the email address, we must write two different functions...

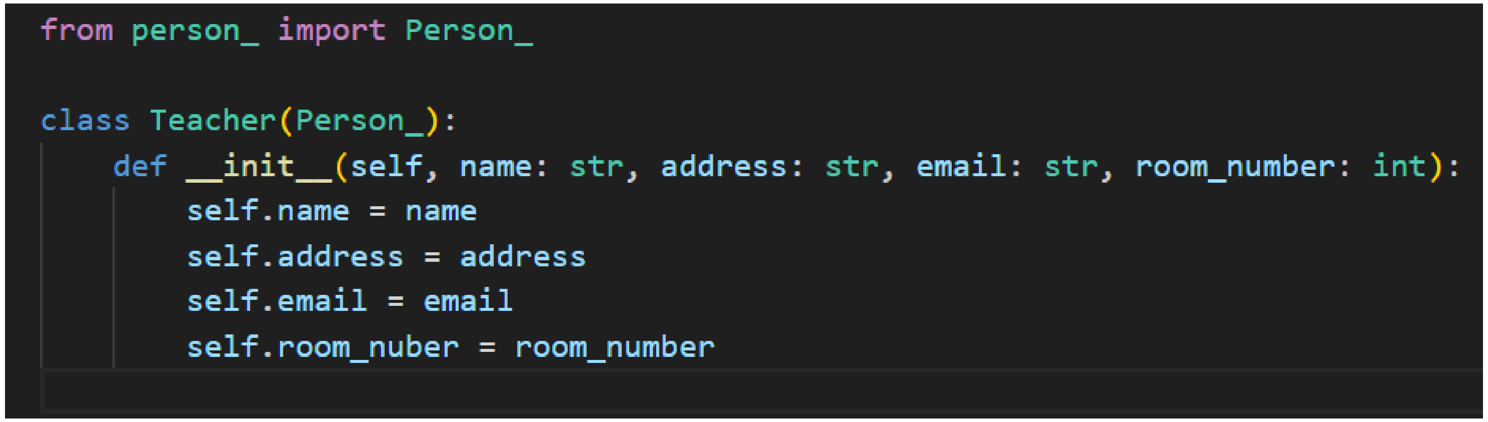


```
Student nbr 1
John Doe
1234@hotmail.fi
john.doe@tylypahka.fi
```

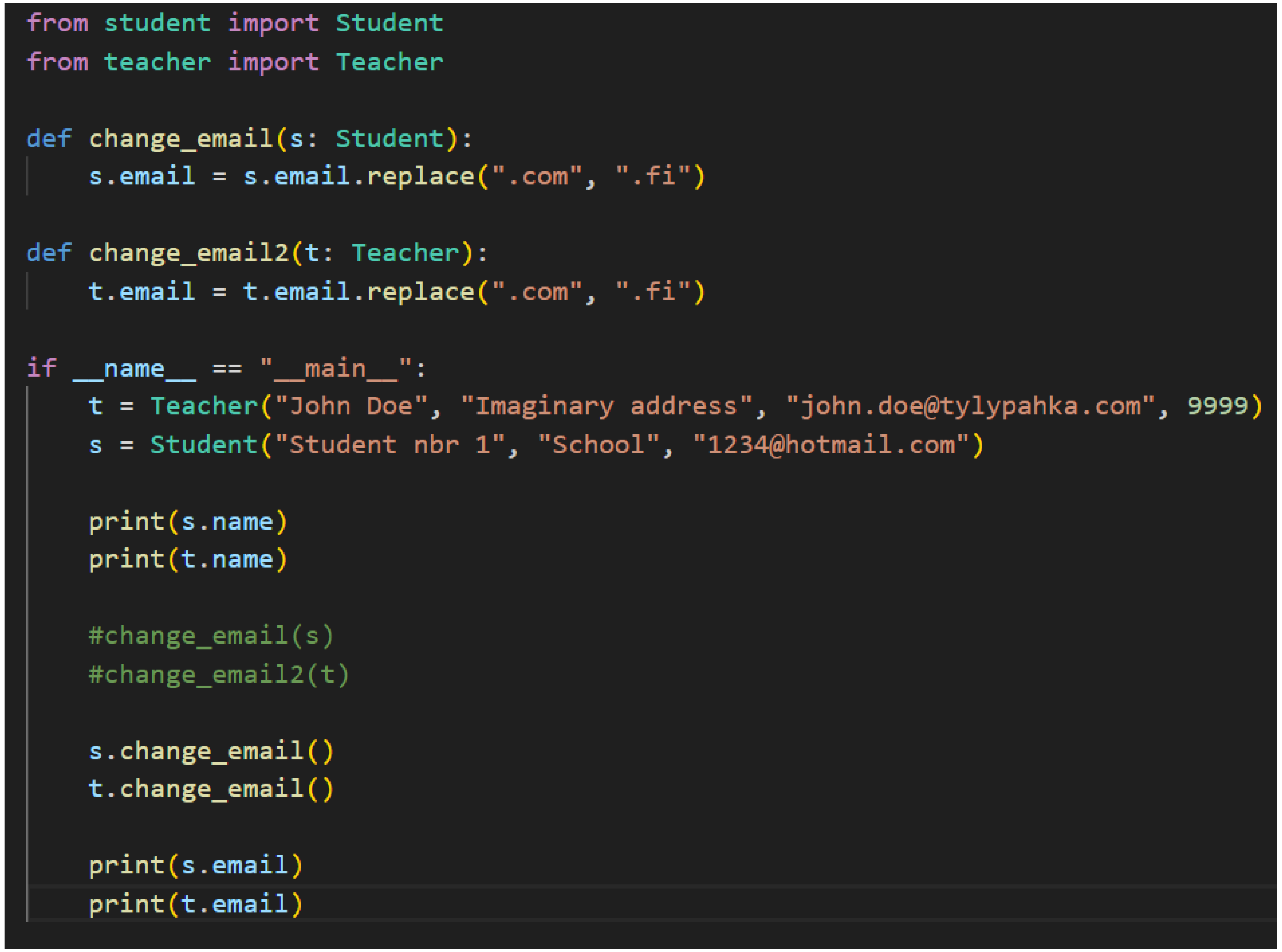
Inheritance:


```python
class Person_:
    def __init__(self, name: str, address: str, email: str):
        self.name = name
        self.address = address
        self.email = email

    def change_email(self):
        self.email = self.email.replace(".com", ".fi")
```

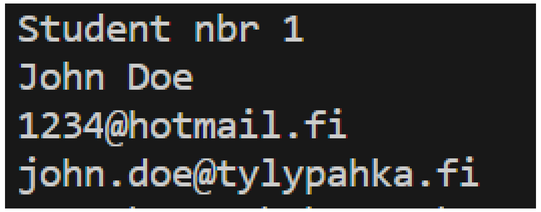

```python
from person_ import Person_

class Student(Person_):
    def __init__(self, name: str, address: str, email: str):
        self.name = name
        self.address = address
        self.email = email
```


```python
from person_ import Person_

class Teacher(Person_):
    def __init__(self, name: str, address: str, email: str, room_number: int):
        self.name = name
        self.address = address
        self.email = email
        self.room_nuber = room_number
```

Using the inherited class change_email call



```python
from student import Student
from teacher import Teacher

def change_email(s: Student):
    s.email = s.email.replace(".com", ".fi")

def change_email2(t: Teacher):
    t.email = t.email.replace(".com", ".fi")

if __name__ == "__main__":
    t = Teacher("John Doe", "Imaginary address", "john.doe@tylypahka.com", 9999)
    s = Student("Student nbr 1", "School", "1234@hotmail.com")

    print(s.name)
    print(t.name)

    #change_email(s)
    #change_email2(t)

    s.change_email()
    t.change_email()

    print(s.email)
    print(t.email)
```

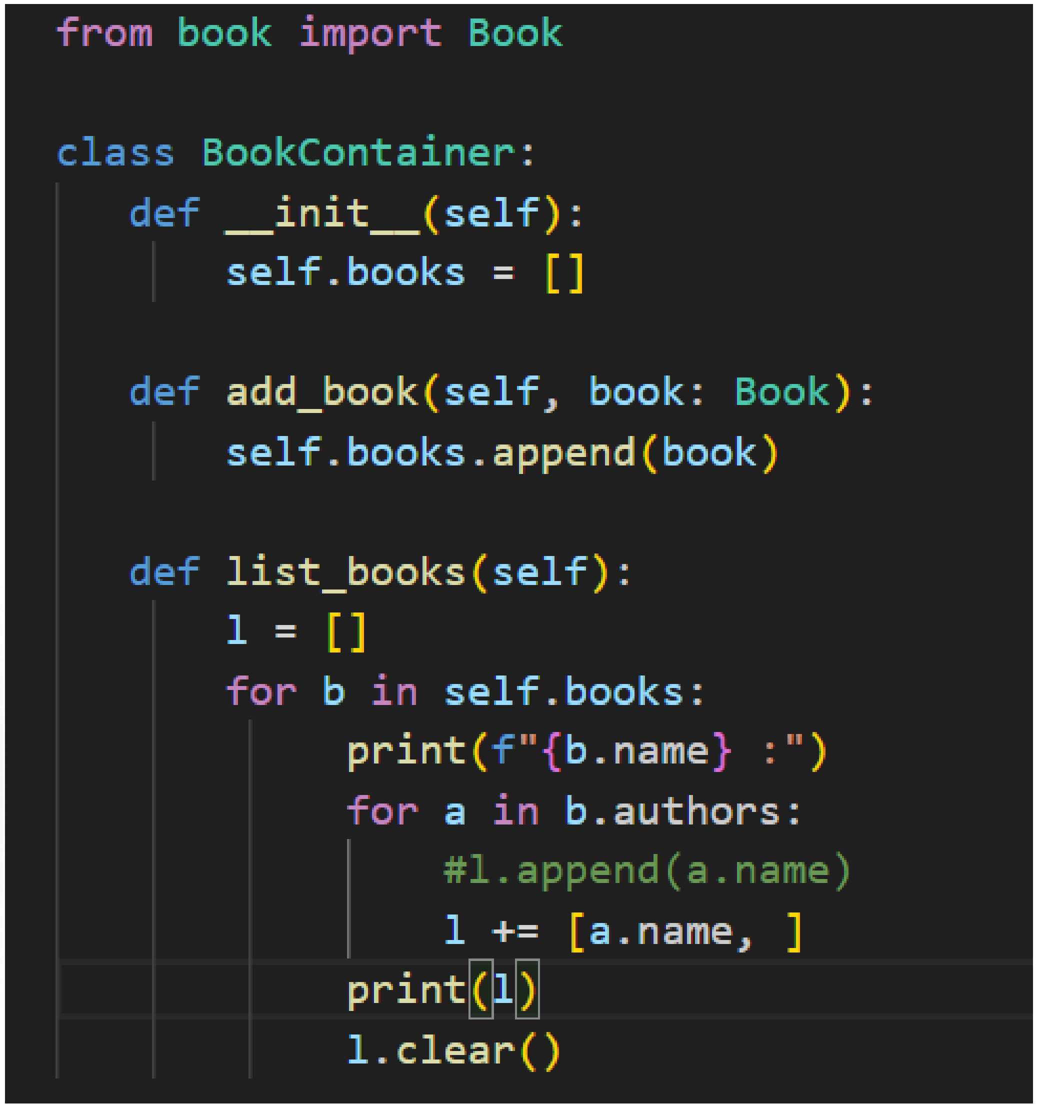


```
Student nbr 1
John Doe
1234@hotmail.fi
john.doe@tylypahka.fi
```

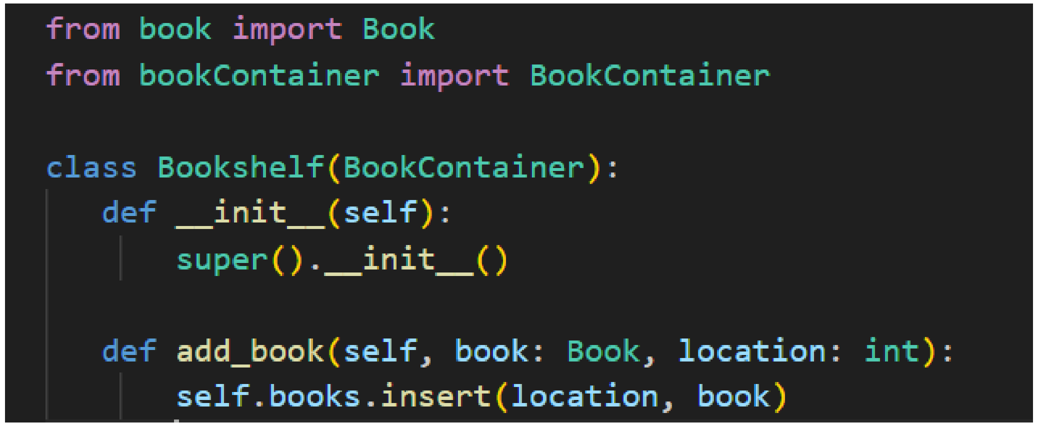
BookContainer & BookShelf – Method overriding

- In the Bookshelf class, there is a method named add_book. The base class BookContainer also defines a method with the same name. This phenomenon is **known as method overriding**. When a derived class has a method sharing the same name as the one in the base class, the version in the derived class takes precedence and replaces the original method when working with instances of the derived class.

- In the example, the concept is that when adding a new book to a BookContainer, it is always placed at the top. However, with a Bookshelf, you have the flexibility to specify the location yourself. The list books method functions identically for both classes, as there is no overridden method in the derived class.



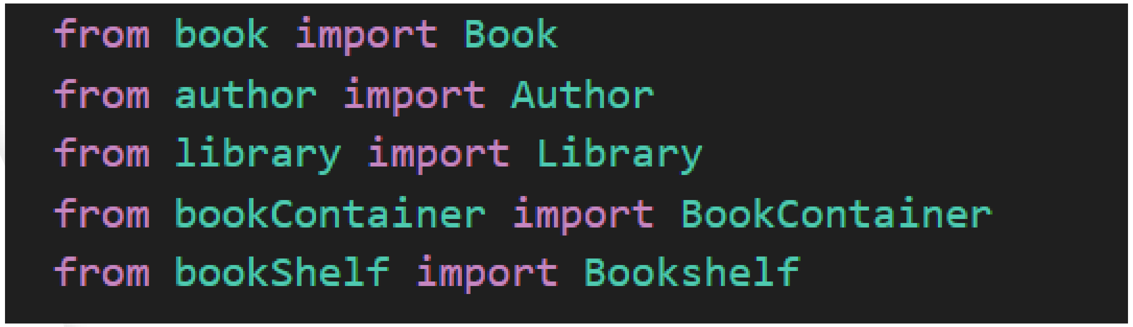
```python
from book import Book

class BookContainer:
    def __init__(self):
        self.books = []

    def add_book(self, book: Book):
        self.books.append(book)

    def list_books(self):
        l = []
        for b in self.books:
            print(f"{b.name} :")
            for a in b.authors:
                #l.append(a.name)
                l += [a.name, ]
            print(l)
            l.clear()
```



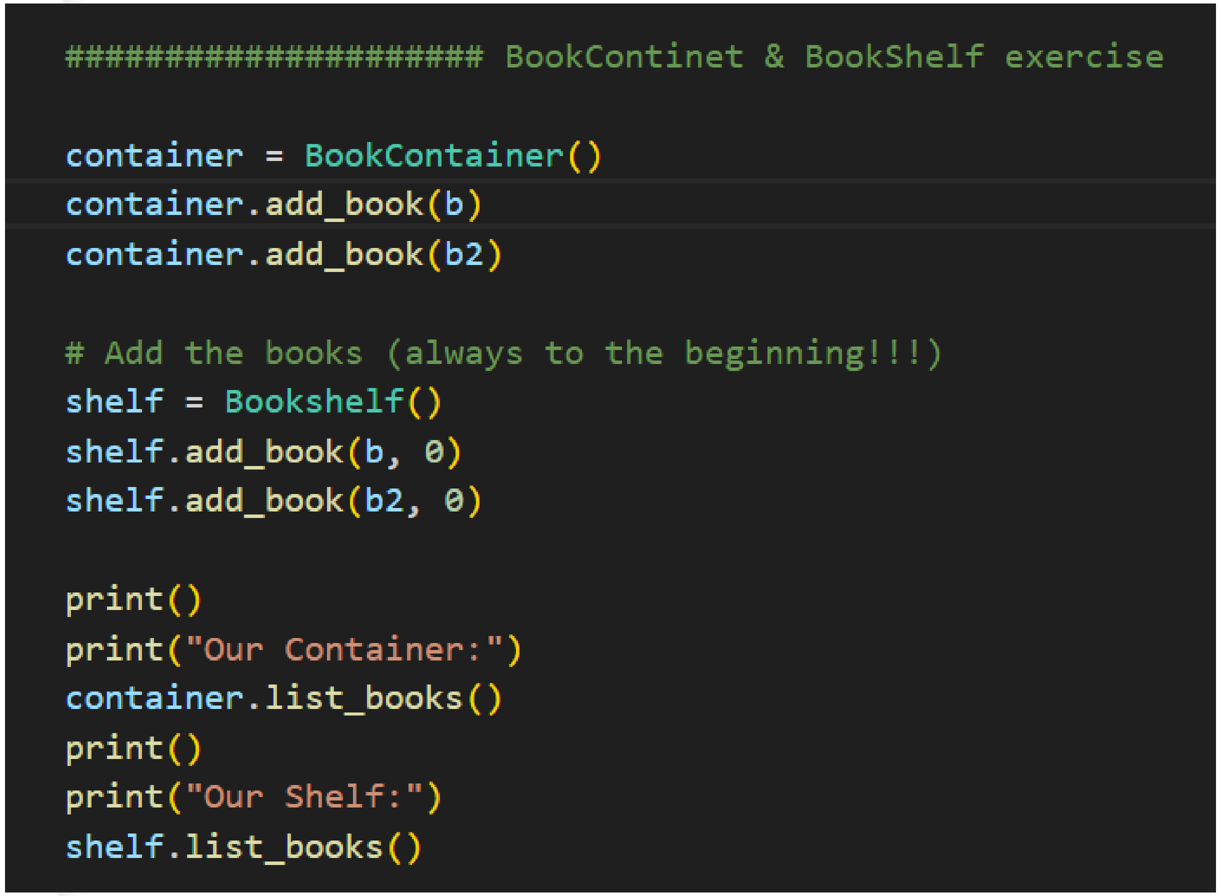
```python
from book import Book
from bookContainer import BookContainer

class Bookshelf(BookContainer):
    def __init__(self):
        super().__init__()

    def add_book(self, book: Book, location: int):
        self.books.insert(location, book)
```

BookContainer & BookShelf in action!!

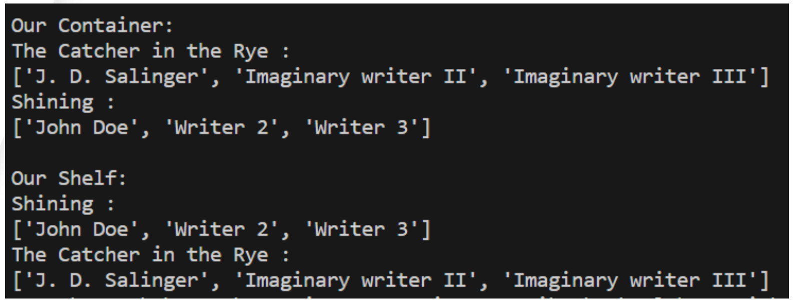
The Bookshelf class also has access to the list of books method. Through inheritance the method is a member of all the classes derived from the BookContainer class!!!!



```
from book import Book
from author import Author
from library import Library
from bookContainer import BookContainer
from bookShelf import Bookshelf
```



```
##################### BookContinet & BookShelf exercise

container = BookContainer()
container.add_book(b)
container.add_book(b2)

# Add the books (always to the beginning!!!)
shelf = Bookshelf()
shelf.add_book(b, 0)
shelf.add_book(b2, 0)

print()
print("Our Container:")
container.list_books()
print()
print("Our Shelf:")
shelf.list_books()
```

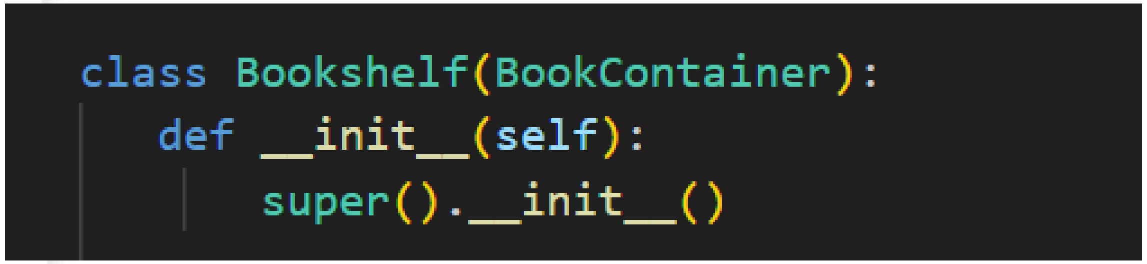


```
Our Container:
The Catcher in the Rye :
['J. D. Salinger', 'Imaginary writer II', 'Imaginary writer III']
Shining :
['John Doe', 'Writer 2', 'Writer 3']

Our Shelf:
Shining :
['John Doe', 'Writer 2', 'Writer 3']
The Catcher in the Rye :
['J. D. Salinger', 'Imaginary writer II', 'Imaginary writer III']
```

Inheritance and the scope of these properties

A derived class inherits all properties from the base class. Properties are accessible from the derived class, unless they have been **defined as private** in the base class (with **two underscores before the name** of the properties).

As the attributes of a Bookshelf are identical to a BookContainer, there was no need to rewrite the constructor of Bookshelf. We simply called the constructor of the base class:



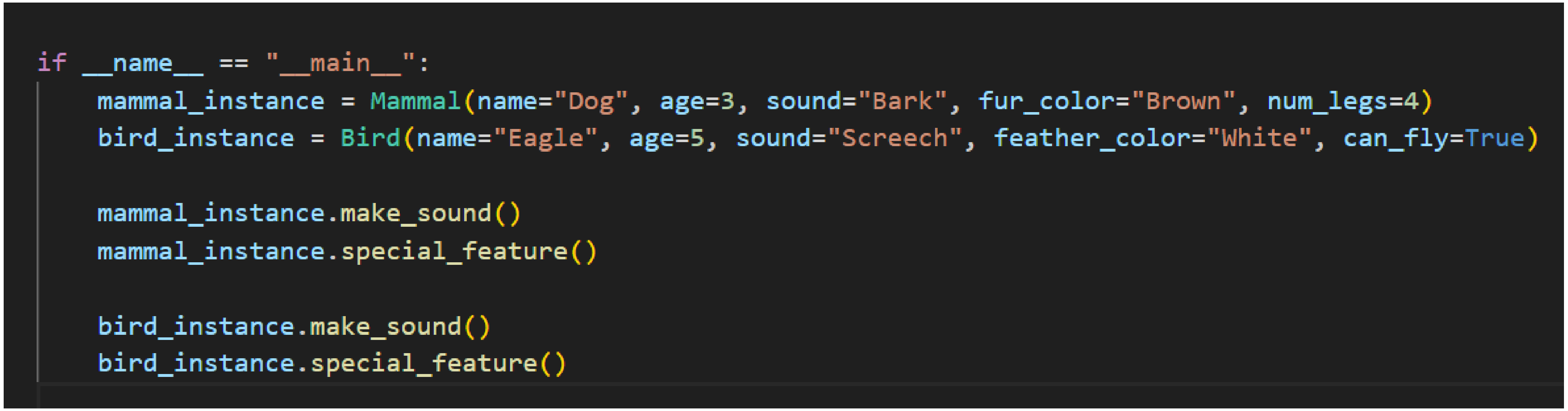
```
class Bookshelf(BookContainer):
    def __init__(self):
        super().__init__()
```

All the properties in the base class can be accessed from the derived class with the function **super()**. The self argument is left out from the method call, as Python adds it automatically.

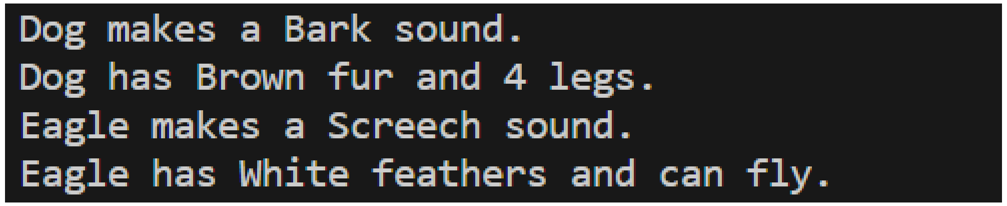
What if the attributes are not identical: can we still use the base class constructor in some way?

Exercise: Inheritance

- Create a Base Class: **Animal**
- Create Subclasses: **Mammal** and **Bird**

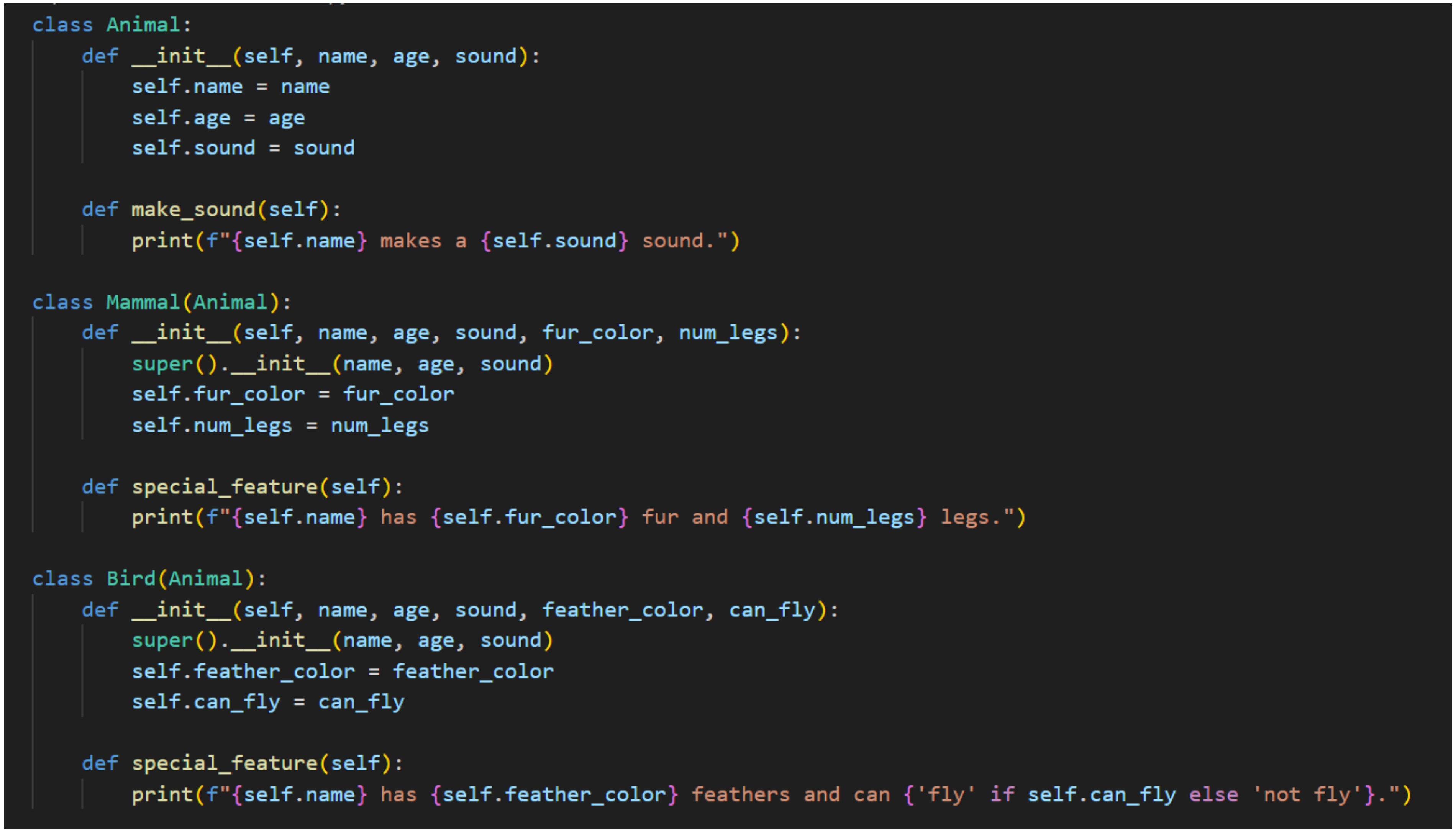


```python
if __name__ == "__main__":
    mammal_instance = Mammal(name="Dog", age=3, sound="Bark", fur_color="Brown", num_legs=4)
    bird_instance = Bird(name="Eagle", age=5, sound="Screech", feather_color="White", can_fly=True)

    mammal_instance.make_sound()
    mammal_instance.special_feature()

    bird_instance.make_sound()
    bird_instance.special_feature()
```



```
Dog makes a Bark sound.
Dog has Brown fur and 4 legs.
Eagle makes a Screech sound.
Eagle has White feathers and can fly.
```

Exercise: Inheritance…



```python
class Animal:
    def __init__(self, name, age, sound):
        self.name = name
        self.age = age
        self.sound = sound

    def make_sound(self):
        print(f"{self.name} makes a {self.sound} sound.")

class Mammal(Animal):
    def __init__(self, name, age, sound, fur_color, num_legs):
        super().__init__(name, age, sound)
        self.fur_color = fur_color
        self.num_legs = num_legs

    def special_feature(self):
        print(f"{self.name} has {self.fur_color} fur and {self.num_legs} legs.")

class Bird(Animal):
    def __init__(self, name, age, sound, feather_color, can_fly):
        super().__init__(name, age, sound)
        self.feather_color = feather_color
        self.can_fly = can_fly

    def special_feature(self):
        print(f"{self.name} has {self.feather_color} feathers and can {'fly' if self.can_fly else 'not fly'}.")
```