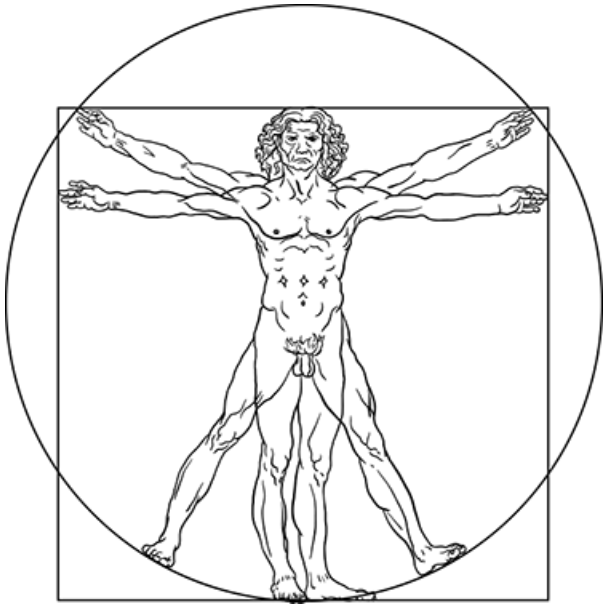
Object-Oriented Programming II

Severin Gsponer
severin.gsponer@insight-centre.org

Last Week...



Class: “abstract”
Blueprint of Data type
Defines attributes and methods

Object: “real”
Instance of a Class
All different and independent

Last week...

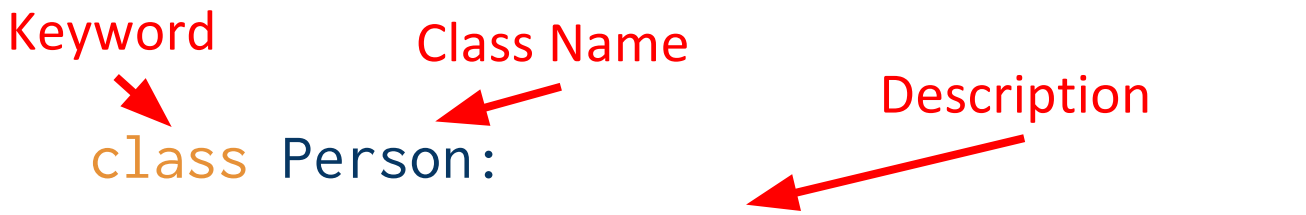
```
class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age

    def become_older(self):
        self.age += 1
```

Last week…




```
class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age

    def become_older(self):
        self.age += 1
```

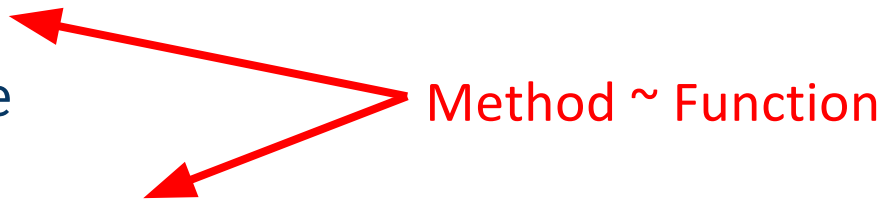
Last week...

```
class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age

    def become_older(self):
        self.age += 1
```



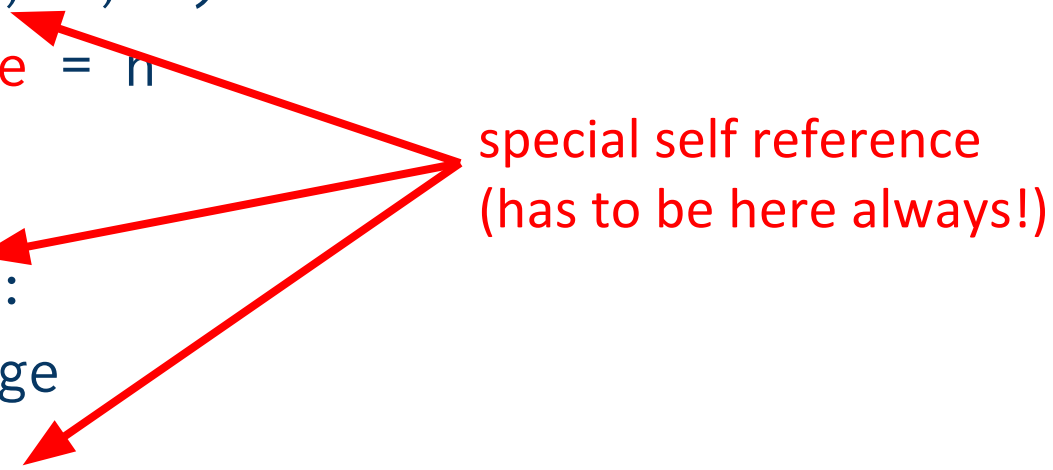
Last week...

```
class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age

    def become_older(self):
        self.age += 1
```



Last week...

```
class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age

    def become_older(self):
        self.age += 1
```

Last week...

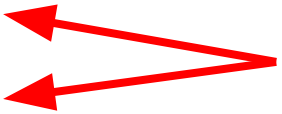
```
from file import Something

class Person:
    """A class representing a Person"""

    def __init__(self, n, a):
        self.full_name = n
        self.age = a
        self.__secrets = []

    def __do_secret_stuff(self):
        # Do something secret
```

module import



Private attribute/method

Last week...

```
from person import Person

persA = Person(“Max”, 23)
persB = Person(“Petra”, 28)

persA.get_age()      # returns 23
persB.get_age()      # returns 28

persA.become_older()

persA.get_age()      # returns 24
persB.get_age()      # returns 28
```

Use dot-notation to access attributes and methods of a Object.

Outline

- Inheritance
- Methods overloading
- Polymorphism
- Special built-in methods and attributes
- Designing classes
- Lab 1 Solution and some additional notes

Extending the Person Class

```
class Person:
    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
    …

class Student(Person):
    def __init__(self, n, a, s):
        Person.__init__(self, n, a)
        self.subject = s

    def get_age(self):
        # Redefines get_age method entirely
        return max(self.age, 18)
```

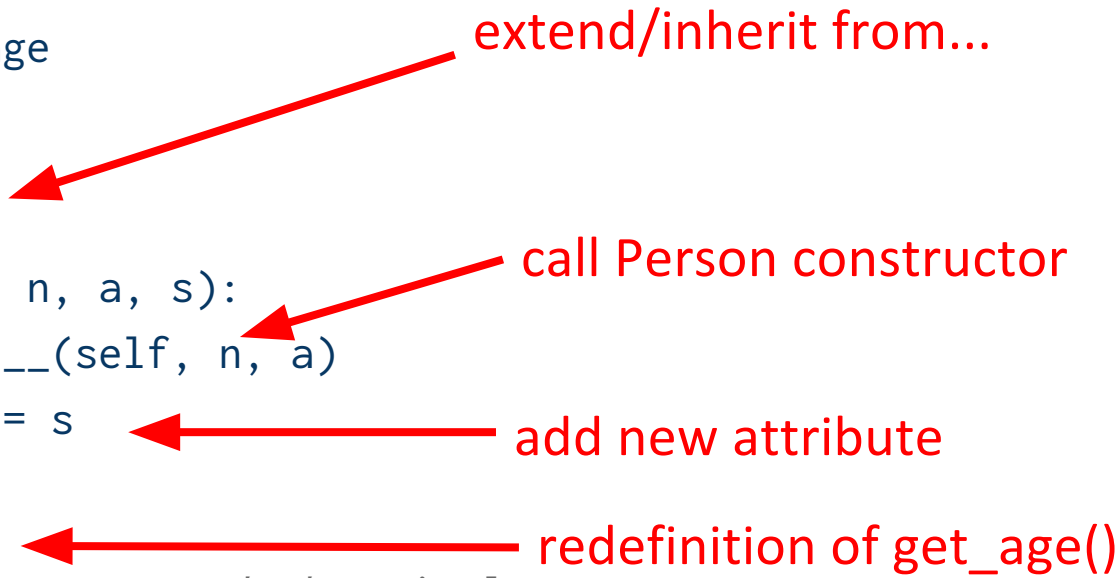
Extending the Person Class

```
class Person:
    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
    …

class Student(Person):
    def __init__(self, n, a, s):
        Person.__init__(self, n, a)
        self.subject = s

    def get_age(self):
        # Redefines get_age method entirely
        return max(self.age, 18)
```



Using the Student class

```
old_student = Student(“Max”, 23, “Cs”)
young_student = Student(“Petra”, 14, “Bio”)

old_student.get_age()      # returns 23
young_student.get_age()    # returns 18

old_student.become_older() # inherited

old_student.get_age()      # returns 25
young_student.get_age()    # returns 18
```

Subclasses

- Classes can **extend** (inherit) the definition of other classes ("is a" relationship)
  - Class get all methods and attributes) of parent class
  - Allows use or extension of methods and attributes already defined in the parent class
- To define a subclass put the name of parent class in parentheses after the class name:

```
class ClassName(ParentClassName):
```

Parent class is often called SuperClass, BaseClass
Child class is often called SubClass, DerivedClass

Add/Overload/Extend Methods

- Add new methods by defining them with a new name

```
class Student(Person):
    def study(self, hours):...
```

- Methods with same name as in the superclass **overloads** (redefines) original method

```
class Student(Person):
    def get_age(self):...
```

- To extend a method of superclass call the method of the superclass explicitly

```
class Student(Person):
    def become_older(self, hours):
        Person.become_older(self)
        self.drink_pints(10)
```

Add/Overload/Extend Methods

- Add new methods by defining them with a new name

```
class Student(Person):
    def study(self, hours):...
```

- Methods with same name as in the superclass **overloads** (redefines) method

```
class Student(Person):
    def get_age(self):...
```

- To extend method of superclass call the method of the superclass explicitly (Syntax: `parentClass.methodName(self)`)

```
class Student(Person):
    def become_older(self, hours):
        Person.become_older(self)
        self.drink_pints(10)
```

only time you ever explicitly pass ‘self’ as an argument

Extending `__init__`

- Same as redefining any other method
  - Very common to call parent's `__init__` method and do additional work (initialization)

```
class Student(Person):
    def __init__(self, n, a, s):
        Person.__init__(self, n, a)
        self.subject = s
```

Polymorphism

- Polymorphism is the ability to present the same interface for different underlying form
  - e.g., “+” “*” are polymorphic operation as they are defined for integers, real numbers, strings, list etc.

- Not that visible in Python as in statically typed languages (e.g., C++, Java)

Polymorphism Example

```
class Animal:
     def __init__(self, name):      # Constructor of the Animal class
          self.name = name
     def talk(self):                # Abstract method, defined by convention only
          raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
     def talk(self):
          return 'Meow!'

class Dog(Animal):
     def talk(self):
          return 'Woof! Woof!'

animals = [Cat('Missy'), Cat('Tigger'), Dog('Luna')]

for animal in animals:
     print animal.name + ': ' + animal.talk()
```

Polymorphism Example

```
class Animal:
     def __init__(self, name):      # Constructor of the Animal class
          self.name = name
     def talk(self):                # Abstract method, defined by convention only
          raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
     def talk(self):
          return 'Meow!'

class Dog(Animal):
     def talk(self):
          return 'Woof! Woof!'

animals = [Cat('Missy'), Cat('Tigger'), Dog('Luna')]

for animal in animals:
     print animal.name + ': ' + animal.talk()
```

Output:

Missy: Meow!

Tigger: Meow!

Luna: Woof! Woof!

The Object Class

- All Classes in Python inherit from the object class

```
class Person(object):
    def __init__(self, n, a):
        self.full_name = n
        self.age = a

    def get_age(self):
        return self.age
```

- Does not need to be written explicitly (Python 3)

Built-in Members of Classes

- Classes contain many methods and attributes that are always included
  - Most define automatic functionality triggered by special operators or usage of that class
  - Built in attributes define information that must be stored for all classes
- All built-in members have double underscores around their names:

  `__init__` `__doc__`

Special Methods

- All the special methods can be redefined

```
class Person(object):
    …
    def __repr__(self):
        return “I’m named “ + self.full_name
    …

pers = Person(“Paul”, 25)
print(pers)                    # Calls (often) pers.__repr__()
“I’m named Paul”               # Exception when __str__ defined
```

In the REPL:

```
>>> pers                       # Calls pers.__repr__()
“I’m named Paul”
```

Special Methods

- You can redefine these and many more as well:

`__init__`	:	The constructor for the class
`__cmp__`	:	Define how == works for class
`__len__`	:	Define how `len(obj)` works
`__copy__`	:	Define how to copy a class

https://docs.python.org/3/reference/datamodel.html#special-method-names

Special Data Item

- These attributes exists for all classes:

  `__doc__` : Documentation string

  `__class__` : Reference to class of the instance

  `__module__` : Reference to module of class

  `__dict__` : Dictionary of of namespace for a class

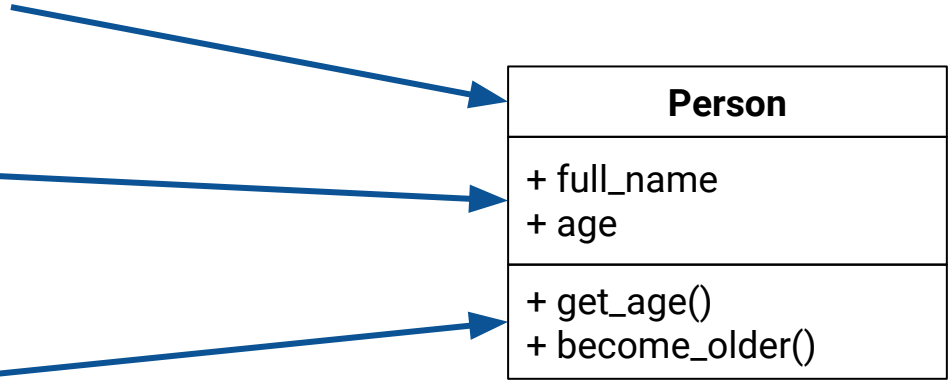
- Useful:
  - `dir(x)` returns a list of all methods and attributes defined for a object `x`

Designing Classes

- The design step is important but difficult task in OOP
  - This is an acquired skill
  - Usually iterative (First draft, improvements, second draft, etc.)
- Some consideration could be:
  - What are the entities involved in the problem
  - What is the responsibility of each entity
  - How these entities interact with each other

Unified Modeling Language - UML

- Name of the class (CamelCase)
- Attributes
- Methods



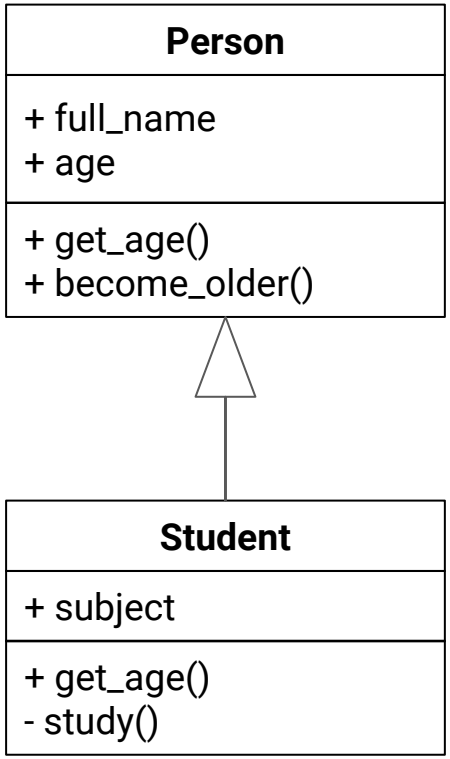
- “+” for public, “-” for private

UML Diagrams - Inheritance

- Inheritance relationship

- Student has three attributes (`full_name`, `age` and `subject`)

- Student has three methods:
  - `get_age()` overloaded
  - `become_older()` inherited
  - `study()` newly added (private)



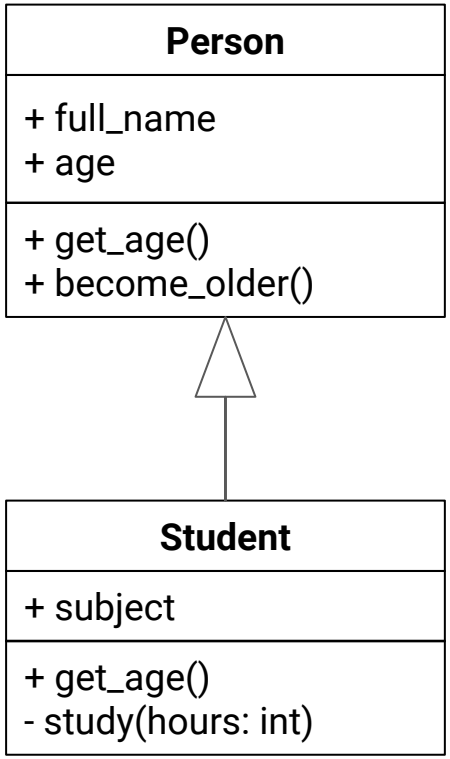
From UML to Code

```
class Person:
      def __init__(self, n, a):
            self.full_name = n
            self.age = a
      def get_age(self):
            return self.age
      def become_older(self): # bla

class Student(Person):
      def __init__(self, n, a, s):
            Person.__init__(self,n , a)
            self.subject = s

      def get_age(self):
            return max(self.age, 18)

      def __study(self, hours):
            # Study hard
```

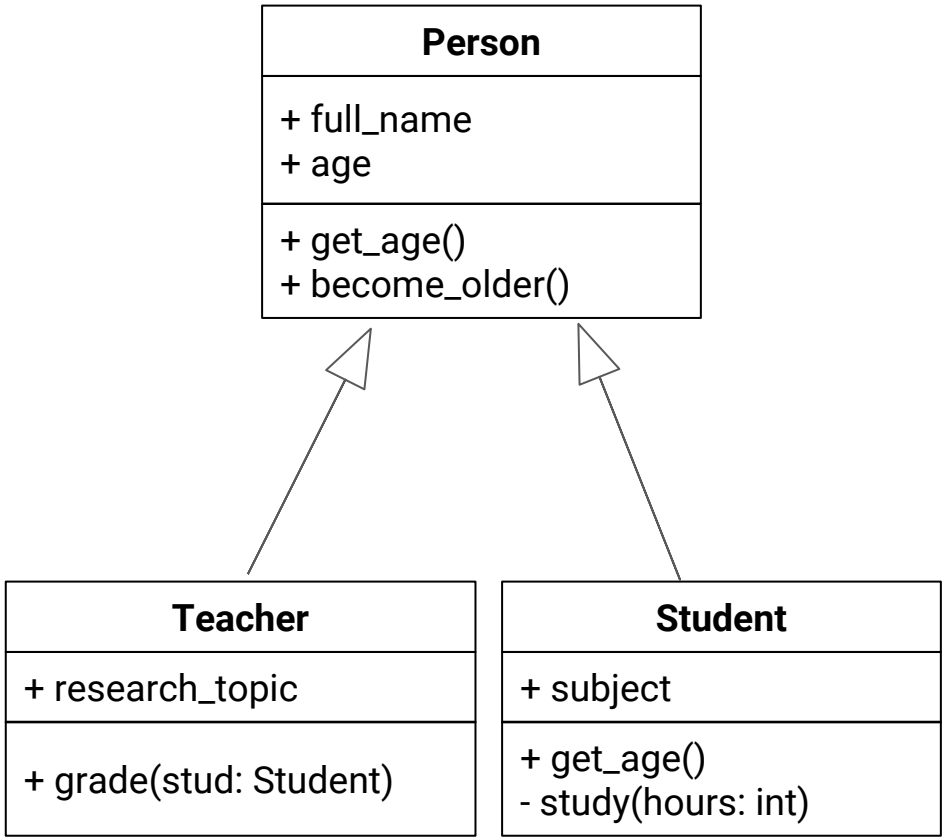


From UML to Code

```
class Person:
      def __init__(self, n, a):
            self.full_name = n
            self.age = a
      def get_age(self): # bla
      def become_older(self): # bla

class Student(Person):
      def __init__(self, n, a, s):
            Person.__init__(self,n , a)
            self.subject = s
      def get_age(self): # bla
      def __study(self, hours): # bla

class Teacher(Person):
      def __init__(self, n, a, s):
            Person.__init__(self,n , a)
            self.research_topic = s
      def grade(self, stud): # bla
```



Conclusion

- Inheritance allows to create hierarchical structure of classes. Overloading and extending methods are useful to specialize more general objects and allow Polymorphism
- UML diagrams are a common way to visualize OOP designs
- Lab this afternoon:
  - Design and build various class structures
  - From UML to code
  - From Text to code
  - From code to UML

More...

https://docs.python.org/3.6/tutorial/classes.html

https://www.youtube.com/watch?v=-DP1i2ZU9gk

https://www.youtube.com/watch?v=FlGjISF3l78

Thanks to Anthony Ventresque and John Tobin for provide most of the slide content.

Solution Lab 1

Notes on Lab 1

- `None` is a special data type (Class)
- Guests are eating, the restaurant only tells them to eat
- Try, fail, and analyse:
  - hard to break stuff
  - `print(x)`, `type(x)`, `help(x)`, `dir(x)`, `x.__dict__`, `len(x)`
  - Read error messages

Environments and Jupyter Notebook

- Create directory for your courses

  Downloads or Home folder might not be the best place
- To start jupyter notebook correctly:

  1. Change to your `courseFolder` (not environment folder)

     ```
     cd <courseFolderName>
     ```

  2. Activate your environment

     ```
     source activate <envname>
     ```

  3. Start jupyter notebook (Do not click on Jupyter Notebook icon)

     ```
     jupyter notebook
     ```