Python Advanced Course

Part I

Stefano Alberto Russo

Outline

- Part I: Object Oriented Programming
    - What is OOP?
    - Logical Example
    - Attributes and methods
    - Why to use objects
    - Defining objects
- Part II: Improving your code
    - Extending objects
    - Lambdas
    - Comprehensions
    - Iterables
    - Properties
- Part III: Exceptions
    - What are exceptions?
    - Handling exceptions
    - Raising exceptions
    - Creating custom exceptions
- Part IV: logging and testing
    - The Python logging module
    - Basics about testing
    - The Python unit-testing module
    - Test-driven development

Outline

- Part I: Object Oriented Programming
  - What is OOP?
  - Logical Example
  - Attributes and methods
  - Why to use objects
  - Defining objects
- Part II: Improving your code
  - Extending objects
  - Lambdas
  - Comprehensions
  - Iterables
  - Properties
- Part III: Exceptions
  - What are exceptions?
  - Handling exceptions
  - Raising exceptions
  - Creating custom exceptions
- Part IV: logging and testing
  - The Python logging module
  - Basics about testing
  - The Python unit-testing module
  - Test-driven development

Object Oriented Programming

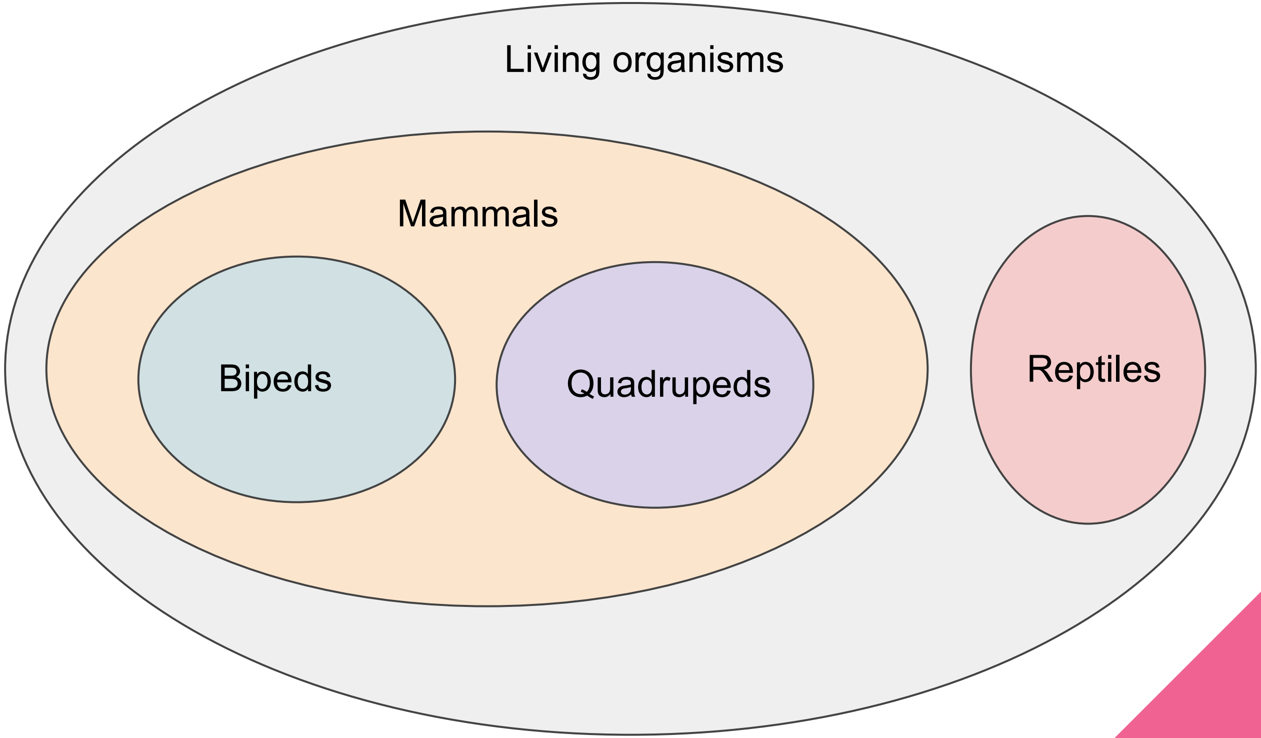
➜ *What is it?*

It is a programming paradigm. Things change quite a lot form “classic” programming. Objects are “entities” which model the world around us.

Objects are defined as *classes*

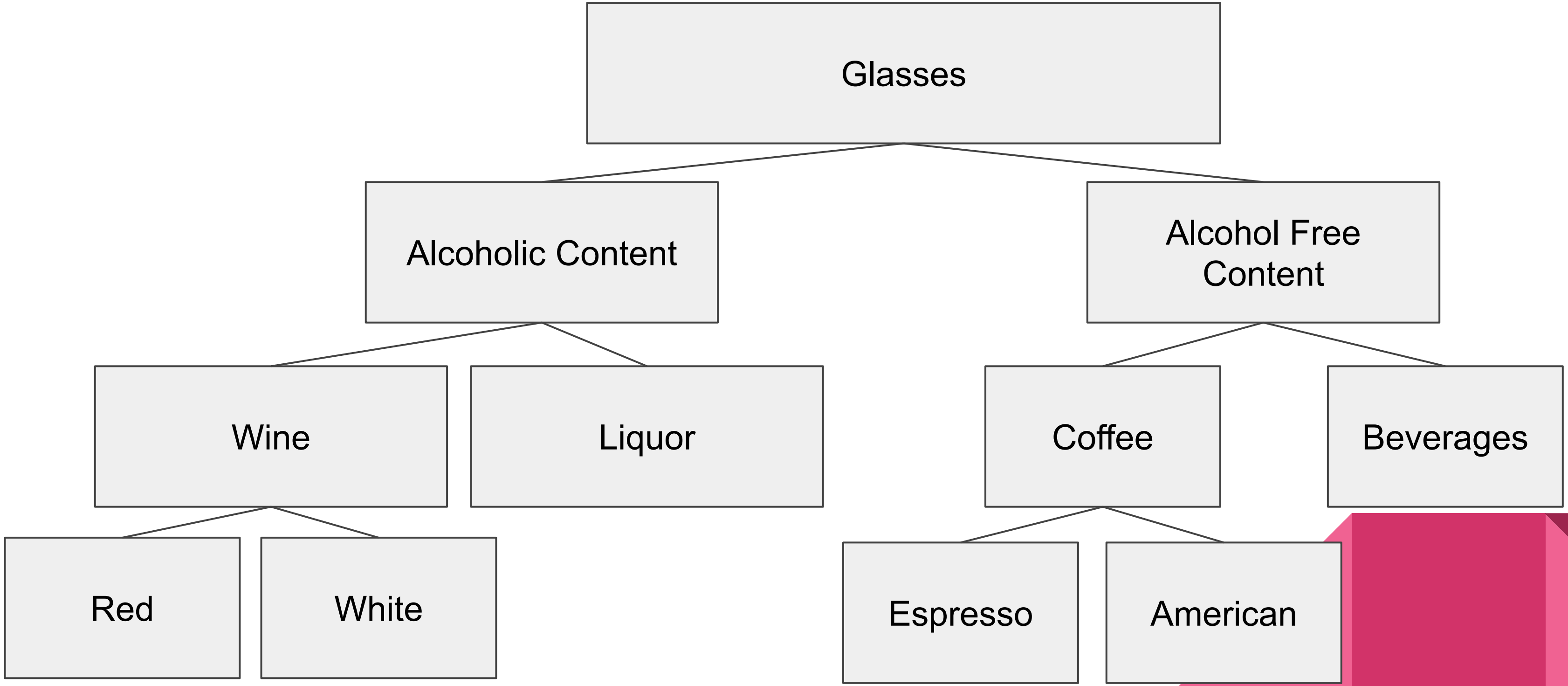
Object Oriented Programming

→ *What is it?*



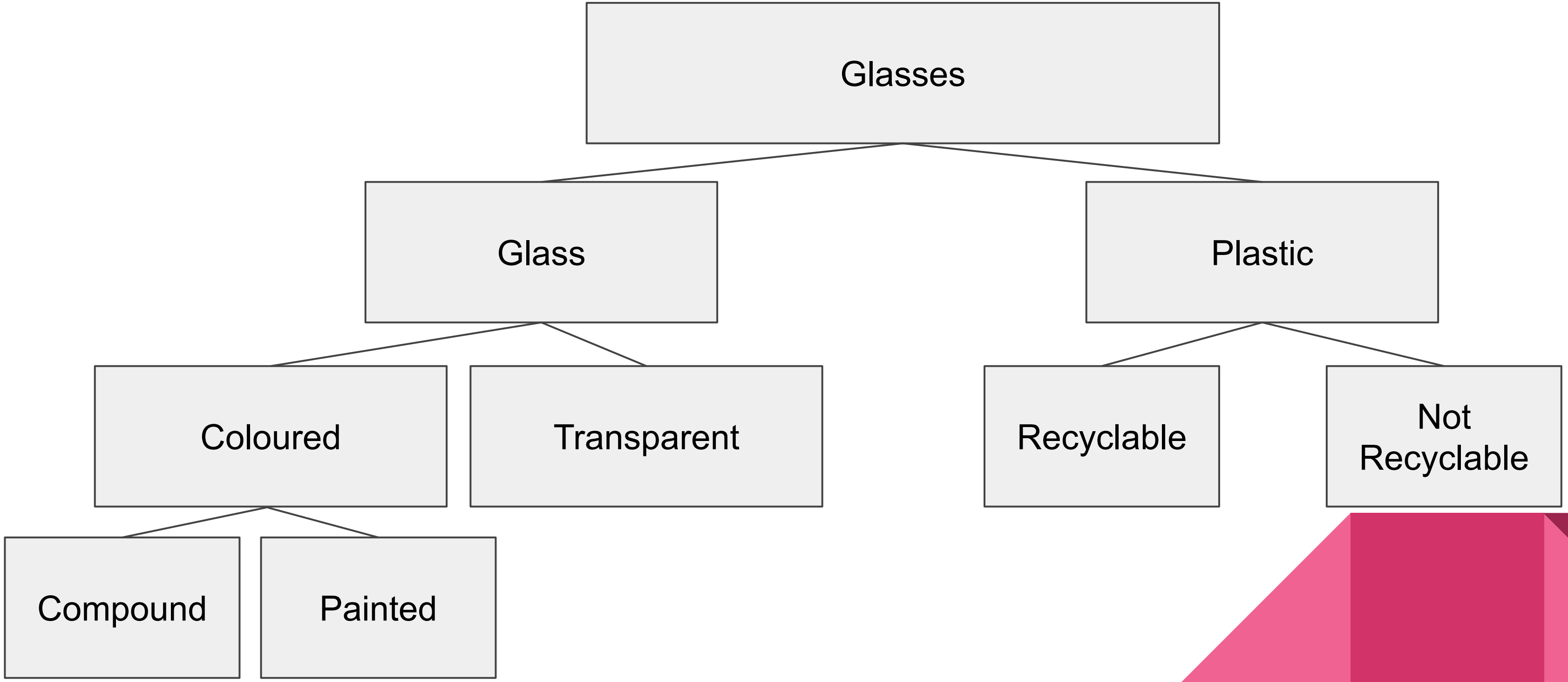
Object Oriented Programming

➔ *What is it?*



Object Oriented Programming

→ *What is it?*



Object Oriented Programming

➔ *What is it?*

It is a programming paradigm. Things change quite a lot form “classic” programming. Objects are “entities” which model the world around us.

Objects are defined as *classes*.

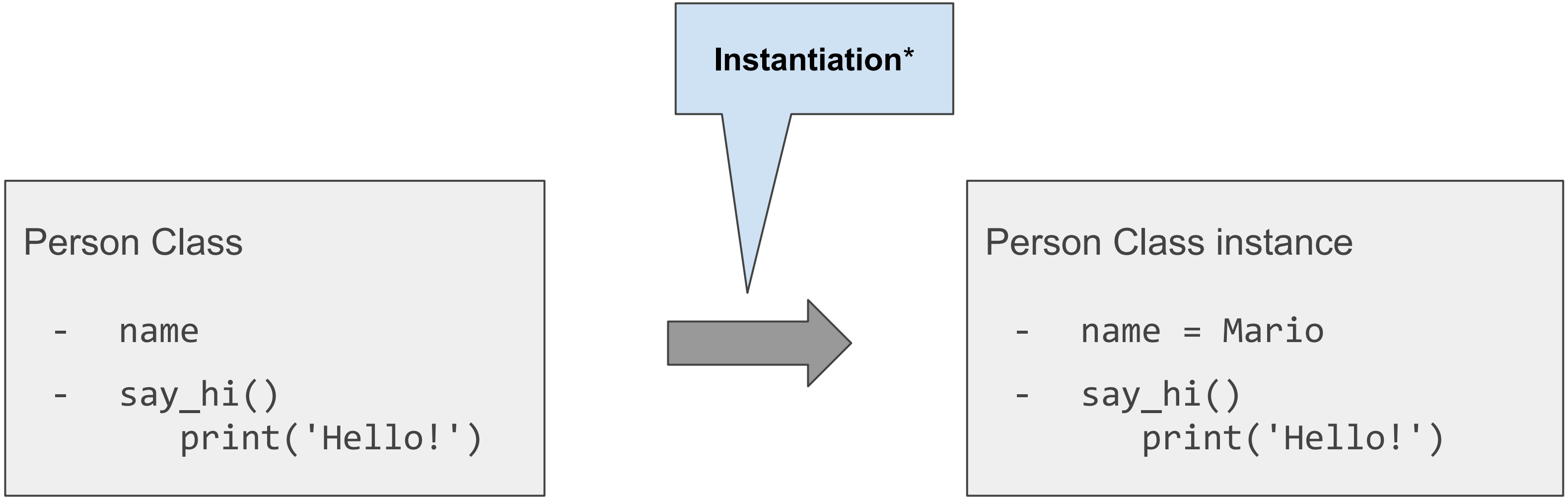
To use objects, we need to create an *instance* of their class.

Objects can have:

- attributes (variables)
- methods (functions)

Object Oriented Programming

→ *Logical Example*



Instantiation*

Person Class

```
-   name
-   say_hi()
        print('Hello!')
```

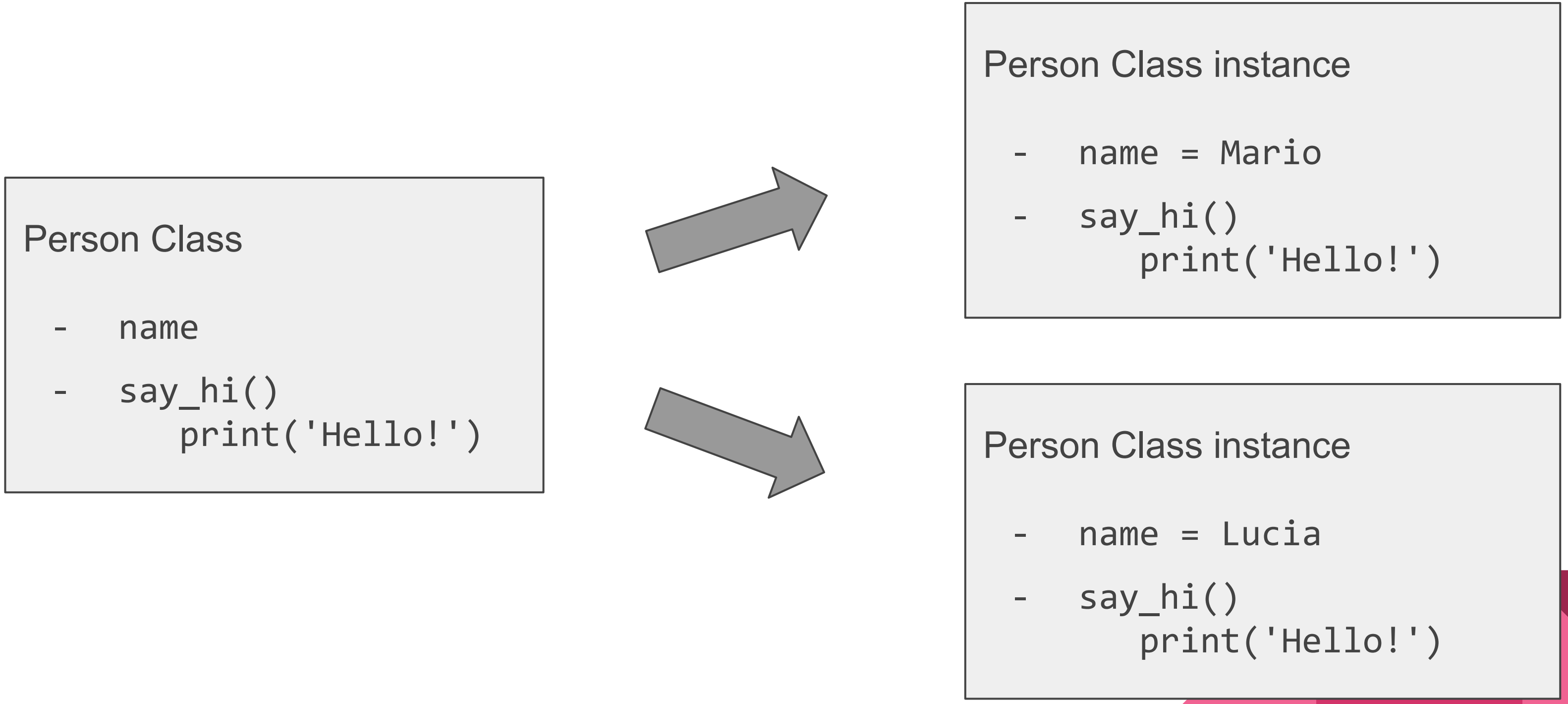
Person Class instance

```
-   name = Mario
-   say_hi()
        print('Hello!')
```

**Also known as construction or initialization*

Object Oriented Programming

→ *Logical Example*



Person Class

```
-   name

-   say_hi()
        print('Hello!')
```

Person Class instance

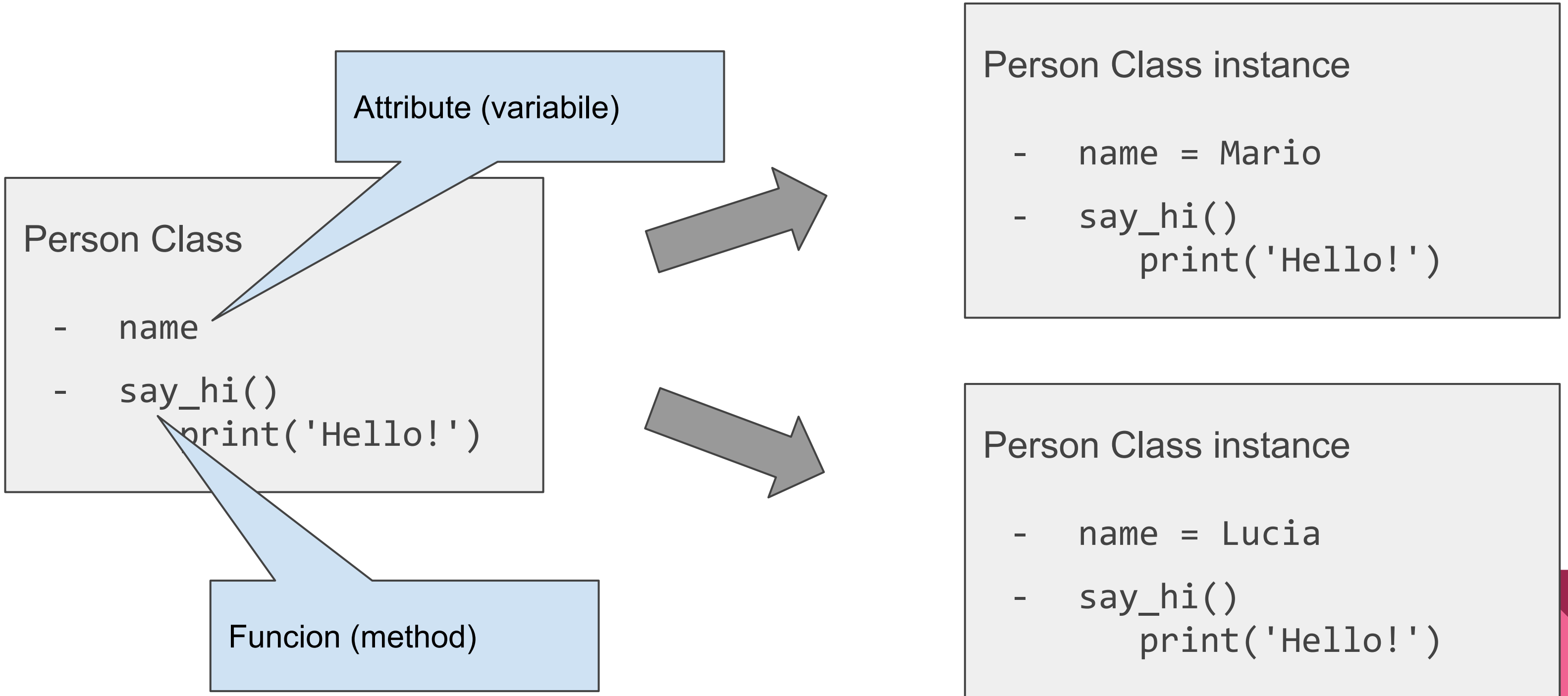
```
-   name = Mario

-   say_hi()
        print('Hello!')
```

Person Class instance

```
-   name = Lucia

-   say_hi()
        print('Hello!')
```

Object Oriented Programming

→ *Logical Example*



Attribute (variabile)

Person Class

```
-   name

-   say_hi()
       print('Hello!')
```

Funcion (method)

Person Class instance

```
-   name = Mario

-   say_hi()
       print('Hello!')
```

Person Class instance

```
-   name = Lucia

-   say_hi()
       print('Hello!')
```

Object Oriented Programming

→ *Class / instance attributes and methods*

By default, attributes and methods depend on the *instance* of the the class: they behave differently for each instance.

However, if they don't have to, then they can be defined as *class* or *static*.

For example, the `say_hi()` function can be be defined as a class method, as it produce the same result regardless of the instance. If instead we wanted to make the `say_hi()` function to include the name of the person, then we couldn't.

Person Class

```
-   name

-   say_hi()
        print('Hello!')
```

Object Oriented Programming

➔ *Why to use objects*

We use object for mainly two reasons:

- The allow to represent vey well hierarchies (and to exploit common characteristics between them)

- Once instantiated, the allow to easily hold the status (without having to rely on external support data structures)

Object Oriented Programming

→ *Conventions*

In Python there is a well defined styling convention:

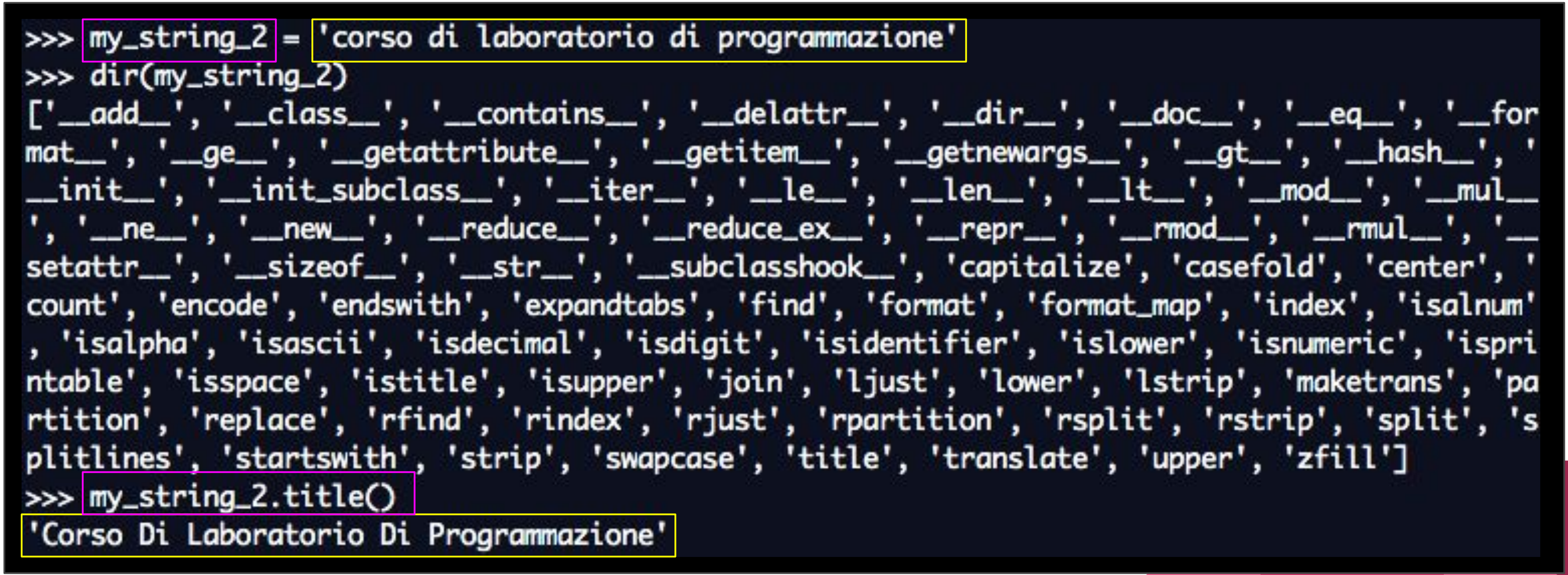
- **lowercase** characters and **underscores** for **variables** and the object **instances**
- **CamelCase** for the **class** names

Moreover, double underscores before and after the name of a method mean that that method is exclusively for internal (private) use, as for the string representation (`__str__`) or the initiator of the object (`__init__`).

→ They are commonly called "magic methods".

Object Oriented Programming

→ *In Python everything is an object*

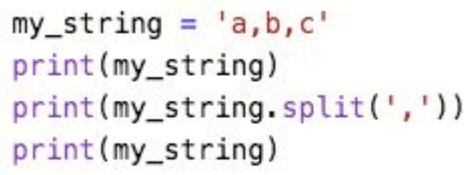


```
>>> my_string_2 = 'corso di laboratorio di programmazione'
>>> dir(my_string_2)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '
__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__
', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__
setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', '
count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum'
, 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'ispri
ntable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'pa
rtition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 's
plitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> my_string_2.title()
'Corso Di Laboratorio Di Programmazione'
```

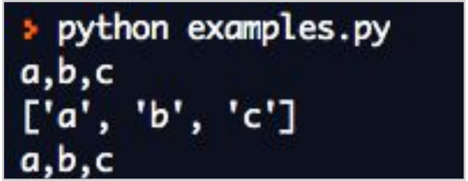
Object Oriented Programming

→ *In Python everything is an object*

examples.py

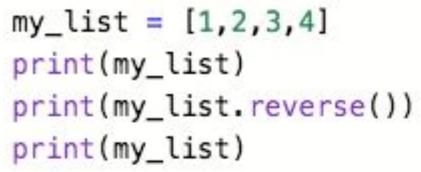


```
my_string = 'a,b,c'
print(my_string)
print(my_string.split(','))
print(my_string)
```

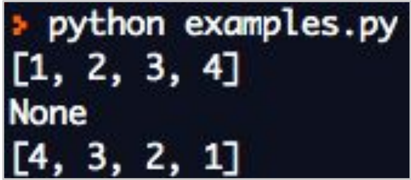


```
> python examples.py
a,b,c
['a', 'b', 'c']
a,b,c
```

examples.py



```
my_list = [1,2,3,4]
print(my_list)
print(my_list.reverse())
print(my_list)
```

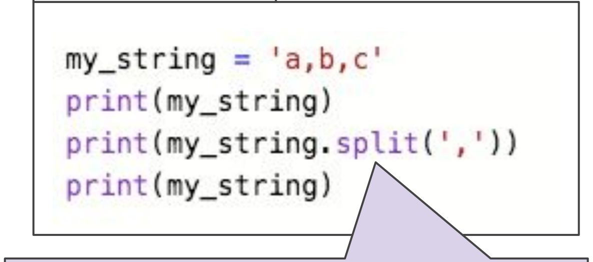


```
> python examples.py
[1, 2, 3, 4]
None
[4, 3, 2, 1]
```

Object Oriented Programming

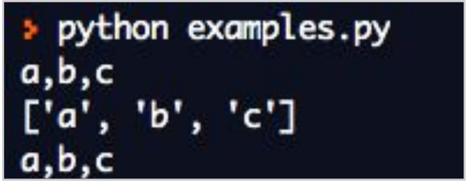
→ *Parenthesis: in-place operations*

examples.py



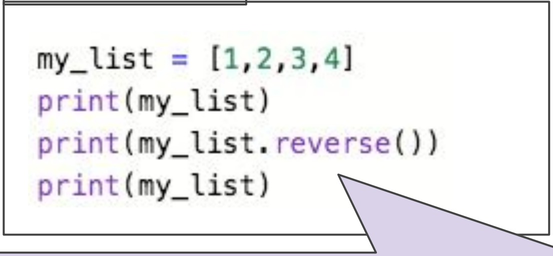
```python
my_string = 'a,b,c'
print(my_string)
print(my_string.split(','))
print(my_string)
```

Operation (function, method) which when executed returns a result



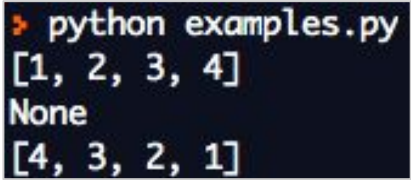
```
> python examples.py
a,b,c
['a', 'b', 'c']
a,b,c
```

examples.py



```python
my_list = [1,2,3,4]
print(my_list)
print(my_list.reverse())
print(my_list)
```

Operation (function, method) which when executed changes the object, does not return anything!

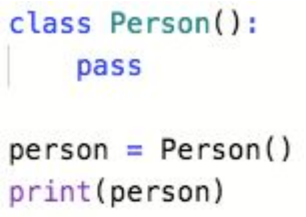


```
> python examples.py
[1, 2, 3, 4]
None
[4, 3, 2, 1]
```

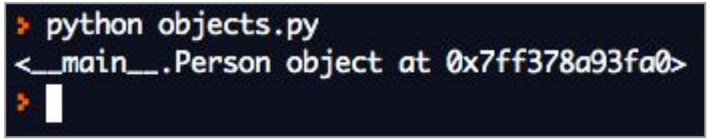
Object Oriented Programming

→ *Defining objects*

objects.py



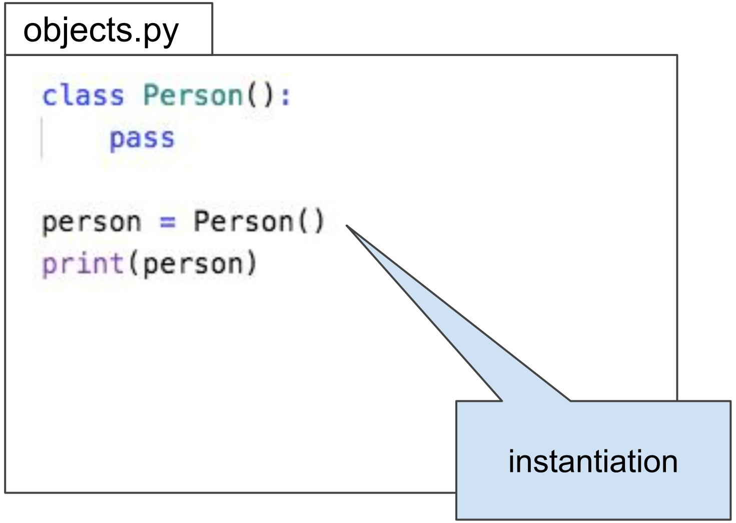
```python
class Person():
    pass

person = Person()
print(person)
```



```
› python objects.py
<__main__.Person object at 0x7ff378a93fa0>
›
```

Object Oriented Programming

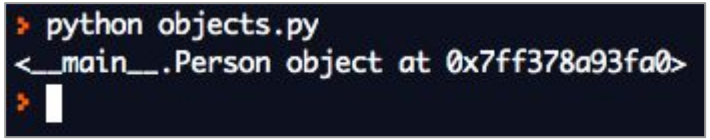
→ *Defining objects*



objects.py

```
class Person():
    pass

person = Person()
print(person)
```

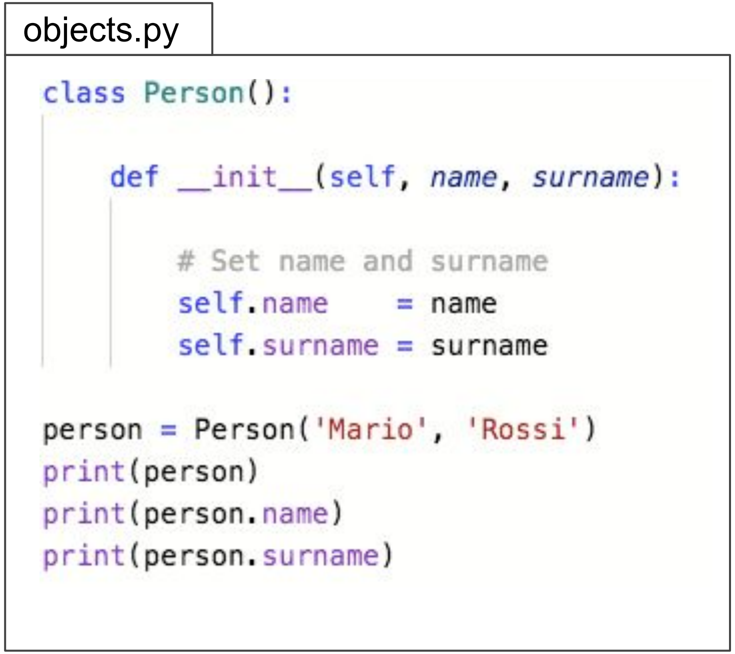
instantiation



```
› python objects.py
<__main__.Person object at 0x7ff378a93fa0>
›
```

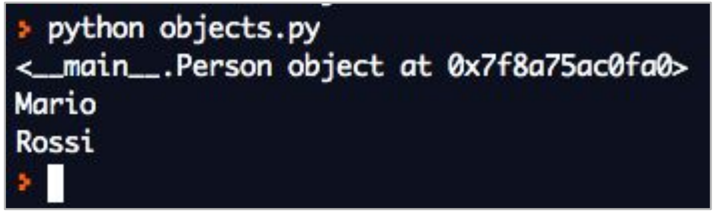
Object Oriented Programming

→ *Defining objects*



objects.py

```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

person = Person('Mario', 'Rossi')
print(person)
print(person.name)
print(person.surname)
```



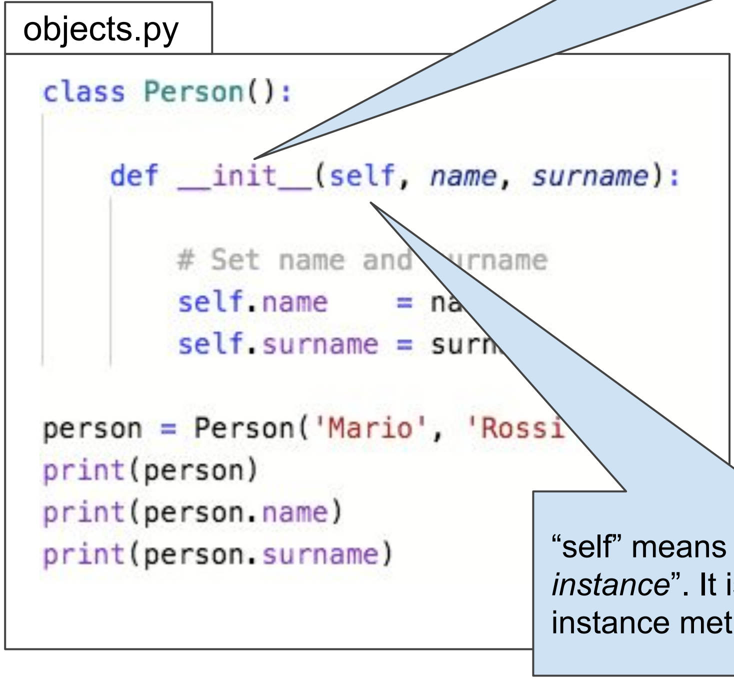
```
> python objects.py
<__main__.Person object at 0x7f8a75ac0fa0>
Mario
Rossi
> 
```

Object Oriented Programming

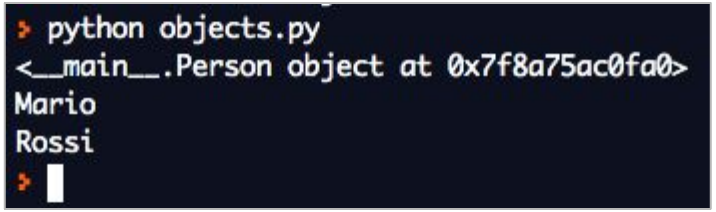
→ *Defining objec*

The “init” function is responsible for initializing the object. If it is not defined, the default one is used, which does nothing.

objects.py



```
class Person():

    def __init__(self, name, surname):

        # Set name and  urname
        self.name    = na
        self.surname = surn

person = Person('Mario', 'Rossi
print(person)
print(person.name)
print(person.surname)
```



```
› python objects.py
<__main__.Person object at 0x7f8a75ac0fa0>
Mario
Rossi
›
```

“self” means “myself”, “myself class *instance*”. It is mandatory in every instance method, even if not used.

Object Oriented Programming

➔ *Defining objects*

- To define class methods, use the ***@classmethod*** decorator. They have the “cls” as first argument instead of the “self”

- To define static methods, use the ***@staticmethod*** decorator. They do not have any special argument (no “self” nor “cls”).

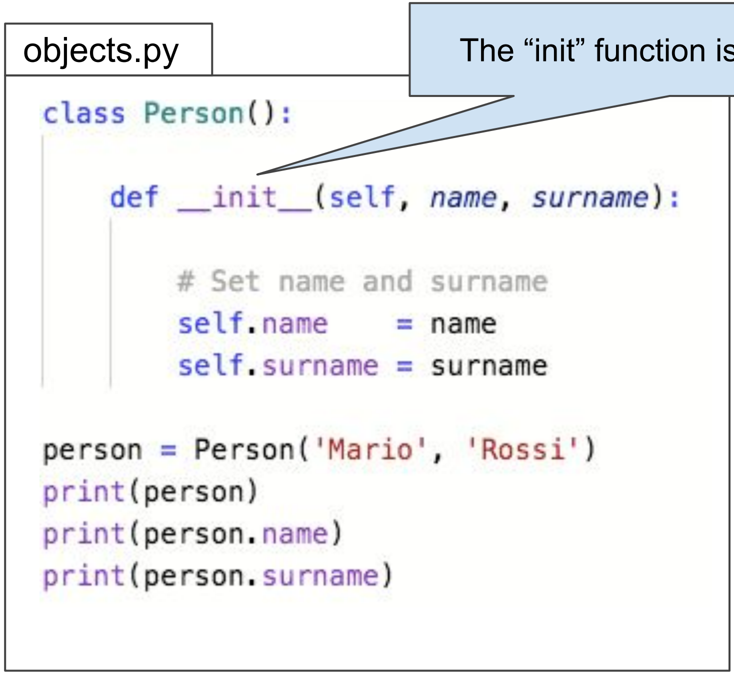
  → A decorator is something placed above a function which “wraps” the function and tells it to behave in a particular way

- To define static/class attributes, define them in the body of the class

Object Oriented Programming

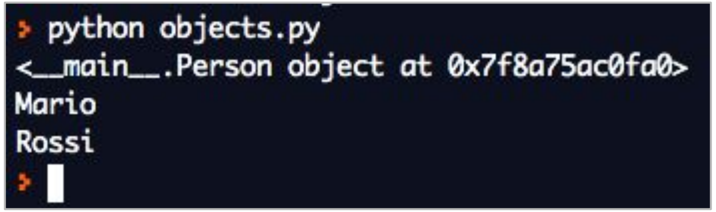
→ *Defining objects*

The “init” function is a magic method.



objects.py

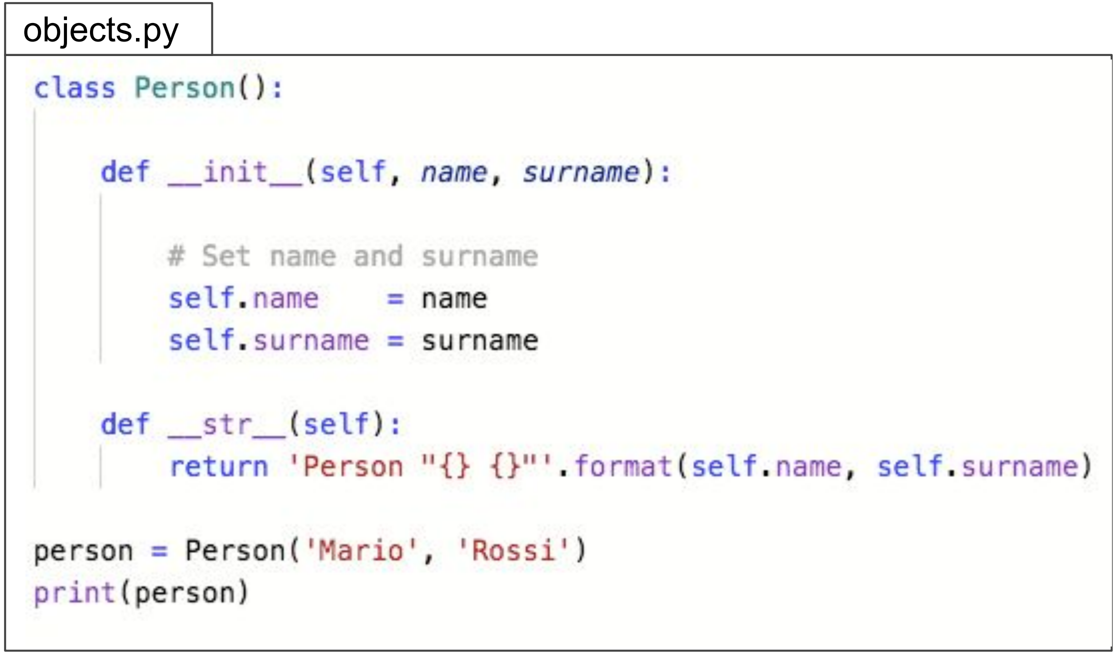
```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

person = Person('Mario', 'Rossi')
print(person)
print(person.name)
print(person.surname)
```



```
> python objects.py
<__main__.Person object at 0x7f8a75ac0fa0>
Mario
Rossi
>
```

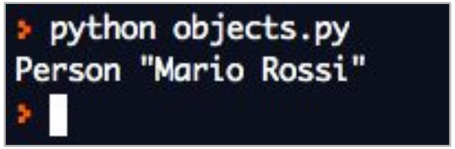
Object Oriented Programming

→ *Magic methods*



objects.py

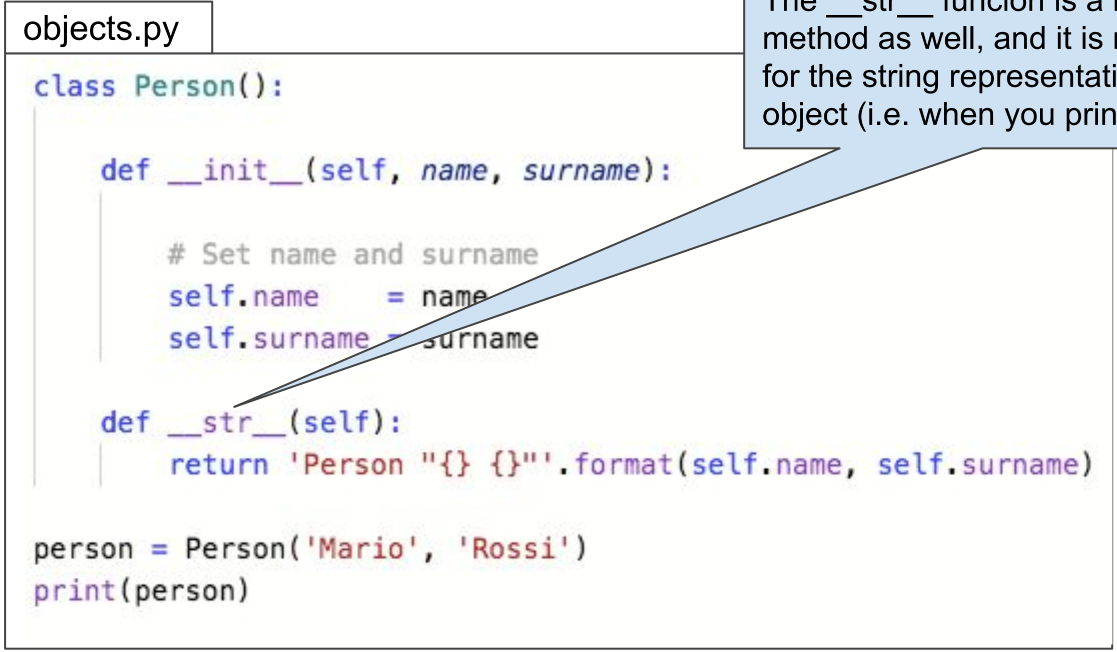
```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

    def __str__(self):
        return 'Person "{} {}"'.format(self.name, self.surname)

person = Person('Mario', 'Rossi')
print(person)
```



```
> python objects.py
Person "Mario Rossi"
> 
```

Object Oriented Programming

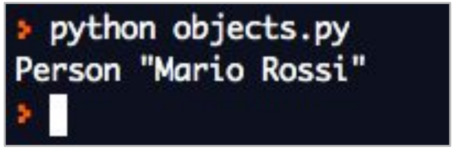
➔ *Magic methods*



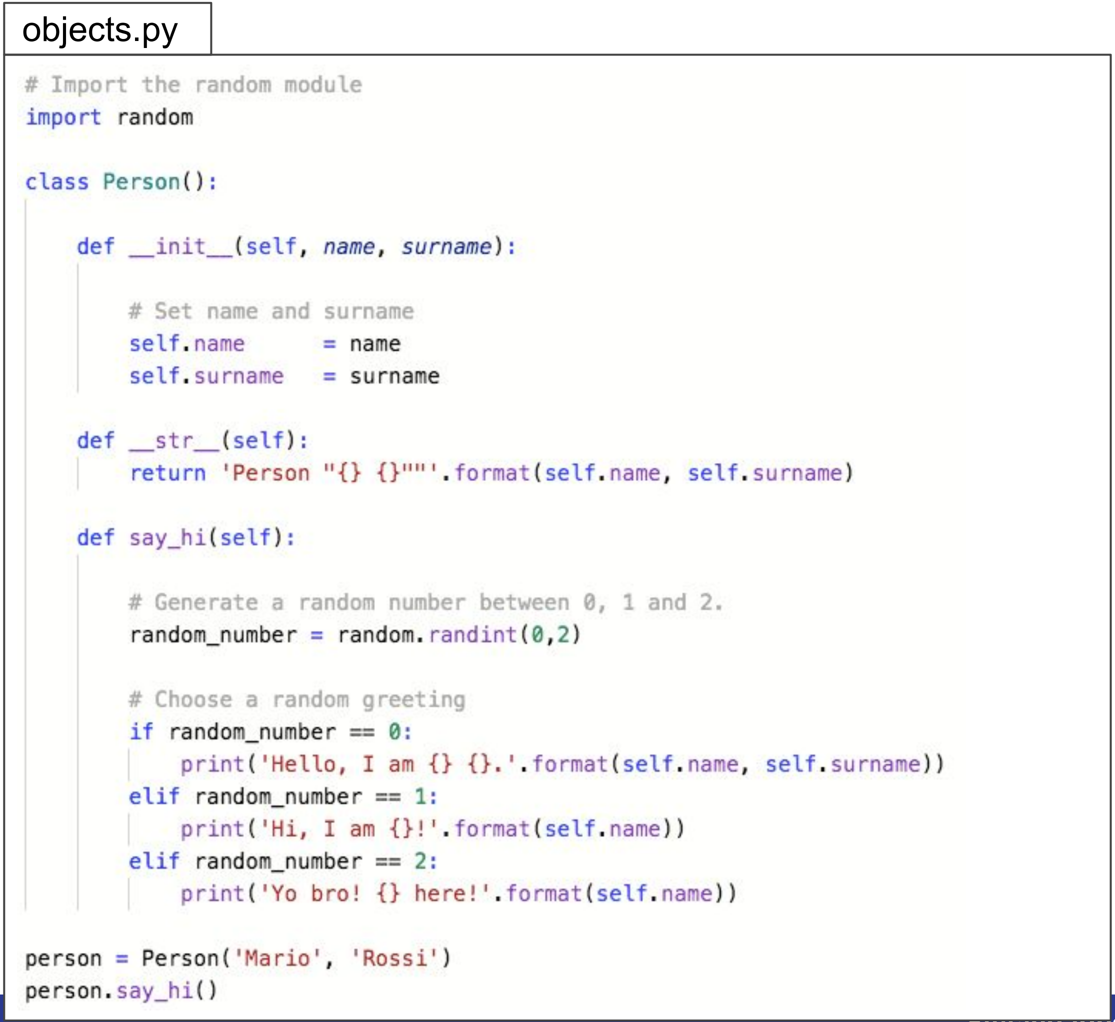
objects.py

```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

    def __str__(self):
        return 'Person "{} {}"'.format(self.name, self.surname)

person = Person('Mario', 'Rossi')
print(person)
```

The __str__ funcion is a magic method as well, and it is responsible for the string representation of the object (i.e. when you print it)

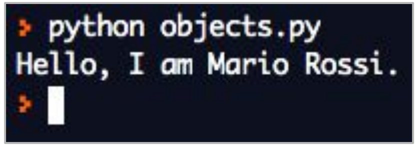


```
› python objects.py
Person "Mario Rossi"
› 
```

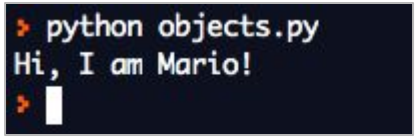


objects.py

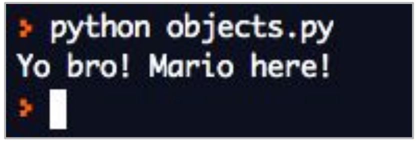
```python
# Import the random module
import random

class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name       = name
        self.surname    = surname

    def __str__(self):
        return 'Person "{} {}""'.format(self.name, self.surname)

    def say_hi(self):

        # Generate a random number between 0, 1 and 2.
        random_number = random.randint(0,2)

        # Choose a random greeting
        if random_number == 0:
            print('Hello, I am {} {}.'.format(self.name, self.surname))
        elif random_number == 1:
            print('Hi, I am {}!'.format(self.name))
        elif random_number == 2:
            print('Yo bro! {} here!'.format(self.name))

person = Person('Mario', 'Rossi')
person.say_hi()
```



```
> python objects.py
Hello, I am Mario Rossi.
>
```

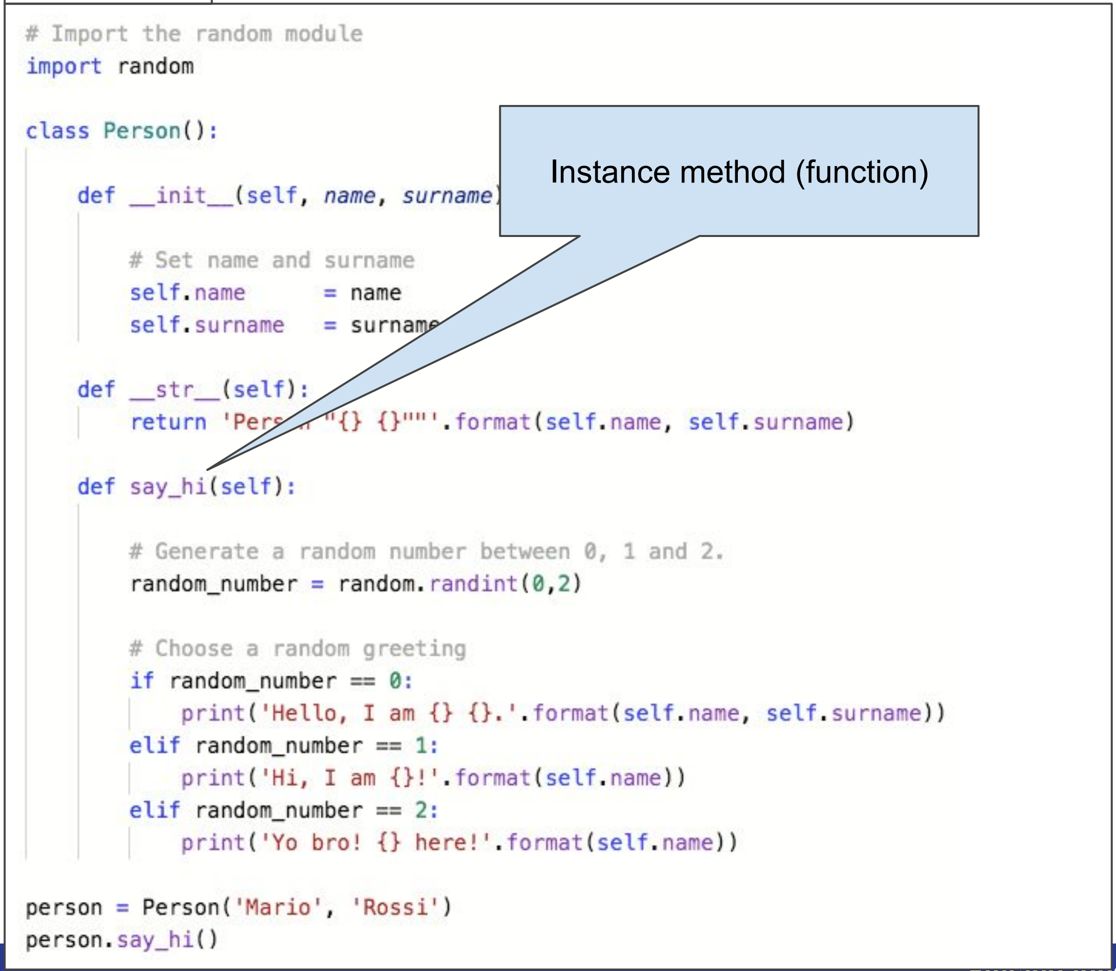


```
> python objects.py
Hi, I am Mario!
>
```



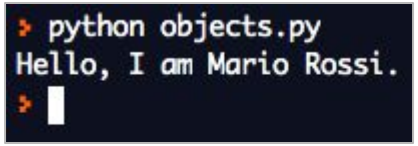
```
> python objects.py
Yo bro! Mario here!
>
```

objects.py

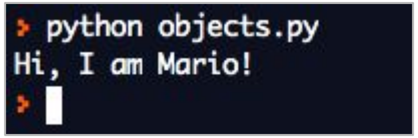


```
# Import the random module
import random

class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name      = name
        self.surname   = surname

    def __str__(self):
        return 'Person "{} {}"""'.format(self.name, self.surname)

    def say_hi(self):

        # Generate a random number between 0, 1 and 2.
        random_number = random.randint(0,2)

        # Choose a random greeting
        if random_number == 0:
            print('Hello, I am {} {}.'.format(self.name, self.surname))
        elif random_number == 1:
            print('Hi, I am {}!'.format(self.name))
        elif random_number == 2:
            print('Yo bro! {} here!'.format(self.name))

person = Person('Mario', 'Rossi')
person.say_hi()
```

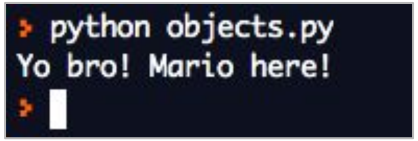
Instance method (function)



```
> python objects.py
Hello, I am Mario Rossi.
>
```



```
> python objects.py
Hi, I am Mario!
>
```



```
> python objects.py
Yo bro! Mario here!
>
```

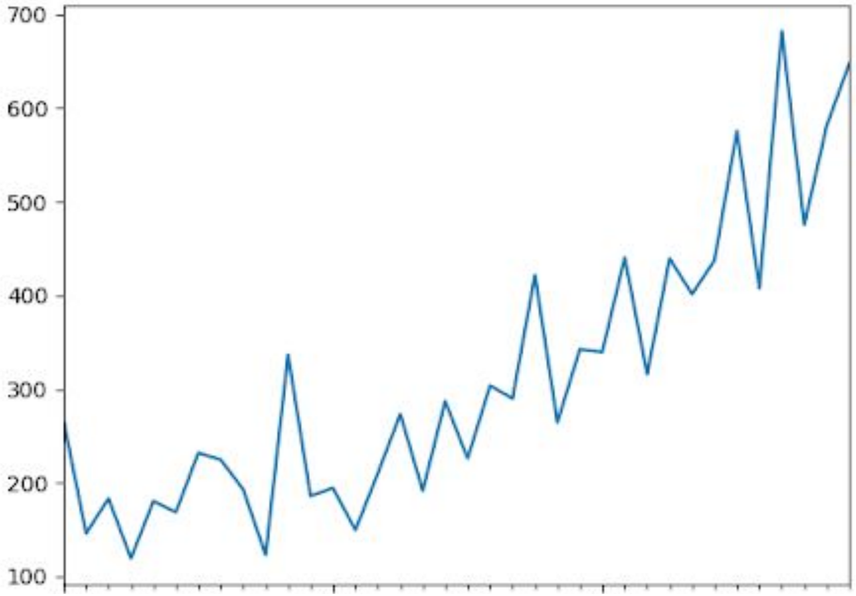
End of part I

- → *Questions?*

Next: exercise 1

Exercise 1

We want to write a predictive model for monthly shampoo sales.



Exercise 1

We want to write a predictive model for monthly shampoo sales.

Our model is extremely simple:

- given a window of **n**
- the sales at **t+1** are given by:
  - the average increment computed over the previous **n** months
  - summed to the last point (**t**) of the window

Exercise 1

→ *Example*

Let’s chose to use 3 months for the prediction (**n=3**) and say that we want to predict the sales for December (**t+1**).

We know that sales for September (**t-2**), October (**t-1**) and November (**t**) have been, respectively, of 50, e 52 e 60 units.

Month	Step	Sales
September	t-2	50
October	t-1	52
November	t (now)	60
December	t+1	?

Exercise 1

→ *Example*

Let’s chose to use 3 months for the prediction (**n=3**) and say that we want to predict the sales for December (**t+1**).

We know that sales for September (**t-2**), October (**t-1**) and November (**t**) have been, respectively, of 50, e 52 e 60 units.

Month	Step	Sales
September	t-2	50
October	t-1	52
November	t (now)	60
December	t+1	**(2+8)/2 + 60 = 65**

Exercise 1

The `IncrementModel()` class must have a *`fit()`* method (which does nothing) and a *`predict()`* method. Both methods must take a “data” argument.

excercise.py

```
class IncrementModel():

    def __init__(self, window)
        self.window = window

    def fit(self, data):
        # Not implemented
        pass

    def predict(self, data):
        # Compute and return the prediction
        prediction = ...
        return prediction
```