

# Python Advanced Course

*Part II*

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 IV: Exceptions
  - What are exceptions?
  - Handling exceptions
  - Raising exceptions
  - Creating custom exceptions
- Part VI: logging and testing
  - The Python logging module
  - Basics about testing
  - The Python unit-testing module
  - Test-driven development

# Improving your code

→ *Extending objects*

objects.py

```
class Person():
    ...

class Student(Person):

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

class Professor(Person):

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

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))
```

# Improving your code

## → *Extending objects*

```
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))
```

# Improving your code

## → Extending objects

objects.py

```
class Person():
    ...

class Student(Person):

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

class Professor(Person):

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

    def say_hi(self):
        print('Hello, I am professor. {} {}'.format(self.name, self.surname))
```

I extend the Person object by declining it into Student and Professor. All the methods that the Person object owned are automatically inherited from the Person and Professor objects. I can overwrite them or add others.

I override the string representation of the Person object to include the title.

I override the greeting method of the Person object to have a more appropriate greeting for a professor.

# Improving your code

→ *Extending objects*

objects.py

```
class Person():
    ...

class Student(Person):

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

class Professor(Person):

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

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

# Improving your code

## → Extending objects

objects.py

```
class Person():
    ...

class Student(Person):

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

class Professor(Person):

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

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

With the "super" I can access the function of the parent object, even if I have overwritten it

# Improving your code

→ *Extending objects*

*Esempio*

objects.py

```
class Person():
    ...

class Student(Person):

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

class Professor(Person):

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

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

```
print('-----')

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

print(person)
person.say_hi()

print('-----')

prof = Professor('Pippo', 'Baudo')

print(prof)
prof.say_hi()
prof.original_say_hi()

print('-----')
```

➤ python objects.py

```
-----
Person "Mario Rossi"
Yo bro! Mario here!
-----
Prof. "Pippo Baudo"
Hello, I am professor Pippo Baudo.
Yo bro! Pippo here!
-----
```



# Improving your code

## → *Reusing code*

- In general, if you write twice the same logic in your code, you should create a support function (or object)
  - Use class / static methods
  - Write external support functions
  - Generalize Objects in parents

# Improving your code

## → *Lambdas*

A lambda function is a small “anonymous” function (not declared)

It can take any number of arguments, but can only have one expression.

Example:

```
x = lambda a : a + 10  
print(x(5))
```

# Improving your code

## → *Lambdas*

They are particularly useful for operating quickly inside other operations,

```
def key(x):  
    return x[1]  
  
a = [(1, 2), (3, 1), (5, 10)]  
a.sort(key=key)
```

v.s.

```
a = [(1, 2), (3, 1), (5, 10)]  
a.sort(key=lambda x: x[1])
```

# Improving your code

## → *List comprehension*

List comprehension allows to quickly create new lists starting from an iterable object (as a list, dictionary, etc).

```
sales_thousands_units = [123.65, 43.67, 124.87]

sales_units = [value*1000 for value in sales_thousands_units]

print(sales_units)
```

```
[123650.0, 43670.0, 124870.0]
```

# Improving your code

## → *List comprehension*

List comprehension allows to quickly create new lists starting from an iterable object (as a list, dictionary, etc).

```
my_dict = {'Venice':10, 'Rome':15}

dict_keys_uppercase = [key.upper() for key in my_dict]

print(dict_keys_uppercase)
```

```
['VENICE', 'ROME']
```

# Improving your code

## → *List comprehension*

It works similarly with dictionaries as well, and a lot of operations get much easier and compact, which improves readability:

```
my_dict = {'Venice': '10', 'Rome': '15'}  
  
new_dict = {int(item):key for key, item in my_dict.items()}  
  
print(new_dict)
```

```
{10: 'Venice', 15: 'Rome'}
```

# Improving your code

## → *Creating iterable objects*

Any object in python can behave as an iterable.

This requires to implement two specific magic methods:

- the `__iter__` to initialize the iteration
- the `__next__` to provide the items for the iteration

# Example

```
class DataSet():  
  
    def __init__(self):  
        self.data = []  
  
    def add(self, item):  
        self.data.append(item)  
  
    def __iter__(self):  
        self.count = 0  
        return self  
  
    def __next__(self):  
        if self.count == len(self.data):  
            raise StopIteration()  
        item = self.data[self.count]  
        self.count += 1  
        return item
```



# Example

```
class DataSet():  
  
    def __init__(self):  
        self.data = []  
  
    def add(self, item):  
        self.data.append(item)  
  
    def __iter__(self):  
        self.count = 0  
        return self  
  
    def __next__(self):  
        if self.count == len(self.data):  
            raise StopIteration()  
        item = self.data[self.count]  
        self.count += 1  
        return item
```

```
data_set = DataSet()  
data_set.add(1)  
data_set.add(2)  
data_set.add(3)  
  
for item in data_set:  
    print(item)
```

# Example

```
class DataSet():  
  
    def __init__(self):  
        self.data = []  
  
    def add(self, item):  
        self.data.append(item)  
  
    def __iter__(self):  
        self.count = 0  
        return self  
  
    def __next__(self):  
        if self.count == len(self.data):  
            raise StopIteration()  
        item = self.data[self.count]  
        self.count += 1  
        return item
```

```
data_set = DataSet()  
data_set.add(1)  
data_set.add(2)  
data_set.add(3)  
  
for item in data_set:  
    print(item)
```

1  
2  
3

# Improving your code

## → *Properties*

Python object can have a special type of attributes, the *properties*.

These are functions which behave as attributes.

Very useful to wrap some logic when accessing /setting an attribute, or to provide alternative view of the data stored inside the object.

# Example


```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def coordinates(self):
        return( (self.x, self.y) )

    @coordinates.setter
    def coordinates(self, coordinates):
        self.x = coordinates[0]
        self.y = coordinates[1]
```

```
point = Point(2,3)
print(point.x)
print(point.y)
print(point.coordinates)

point.coordinates = (4,5)
print(point.x)
print(point.y)
print(point.coordinates)
```



```
2
3
(2, 3)

4
5
(4, 5)
```

# End of part II

→ *Questions?*

**Next: exercise 2**

## Exercise 2

We want to extend our predictive model for monthly shampoo sales.

Now we also want to implement a fit function which computes the average increment over the entire dataset

Our model is extremely simple:

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

# Exercise 2

→ *Example*

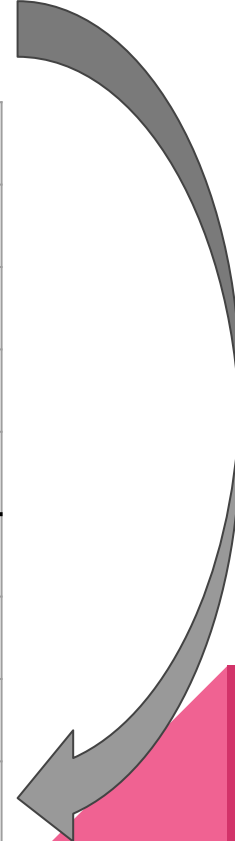
Month	Step	Sales
May	not relevant	8
June	not relevant	19
July	not relevant	31
August	not relevant	41
September	t-2	50
October	t-1	52
November	t (now)	60
December	t+1	?

# Exercise 2

→ *Example*

$$60 + ( ( ( (11+12+10) / 3) + ( (2+8) / 2) ) / 2 )$$

Month	Step	Sales
May	not relevant	8
June	not relevant	19
July	not relevant	31
August	not relevant	41
September	t-2	50
October	t-1	52
November	t (now)	60
December	t+1	68





# Exercise 2

The `FitIncrementModel()` class must have a *fit()* method and a *predict()* method. Both methods must take a “data” argument.

exercice.py

```
class FitIncrementModel():  
  
    def __init__(self, window)  
        self.window = window  
  
    def fit(self, data):  
        # Compute and store the avg hist. increment  
        self.hist_avg_increment = ...  
  
    def predict(self, data):  
        # Compute and return the prediction  
        prediction = ...  
        return prediction
```

# Exercise 2

The `FitIncrementModel()` class must have a *fit()* method and a *predict()* method. Both methods must take a “data” argument.

exercice.py

```
class FitIncrementModel():  
  
    def __init__(self, window)  
        self.window = window  
  
    def fit(self, data):  
        # Compute and store the avg hist. increment  
        self.hist_avg_increment = ...  
  
    def predict(self, data):  
        # Compute and return the prediction  
        prediction = ...  
        return prediction
```

excercise.py

```
class Model():  
    def fit(self, data):  
        pass  
  
    def predict(self, data):  
        pass  
  
class IncrementModel(Model):  
    def fit(self, data):  
        pass  
  
    def predict(self, data):  
        pass  
  
class FitIncrementModel(IncrementModel):  
    def fit(self, data):  
        pass  
  
    def predict(self, data):  
        pass
```