Good coding practices (in Python)

Alberto Tonda, Ph.D., Senior permanent researcher (DR)

UMR 518 Mathématiques et Informatique Appliquées - PS, INRAE, U. Paris-Saclay
UAR 3611 Institut des Systèmes Complexes Paris Ile-de-France, CNRS

alberto.tonda@inrae.fr

Another example of title for a slide

- This is some text
  - And some smaller text

Why is this tricky to explain?

- There are a lot of interdependent pieces
  - Definitions kinda depend on each other
  - Usually I learn a language by trying to do something with it
- We might swing from *trivial* to **extremely difficult** concepts!

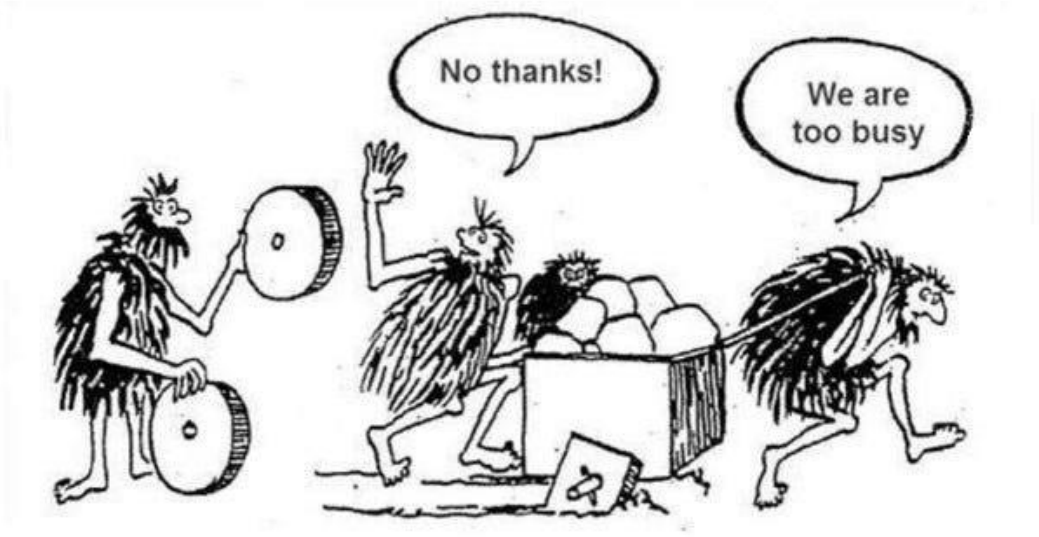
Why Python?

- High level of abstraction
  - Just like MatLab, or R: don’t bother with small details (memory)
  - Easy(-ier) to learn than lower-level languages (C++)
- Widely used in research
  - Open source, seen as an alternative to MatLab
  - Interpreted, same program can run on Windows/Linux/Mac
- Vast amount of libraries
  - Since it’s easy to use, lots of people added their own packages
  - Especially for Machine Learning/Data Science

Libraries, packages, …?

- Collection of code written by someone else
  - Popularity of languages heavily depends on libraries
  - In Python, they are called **packages**
  - Python has a considerable amount of packages!

- Do not reinvent the wheel
  - Check if someone else already did it
  - Learn to use their code



Integrated Development Environment

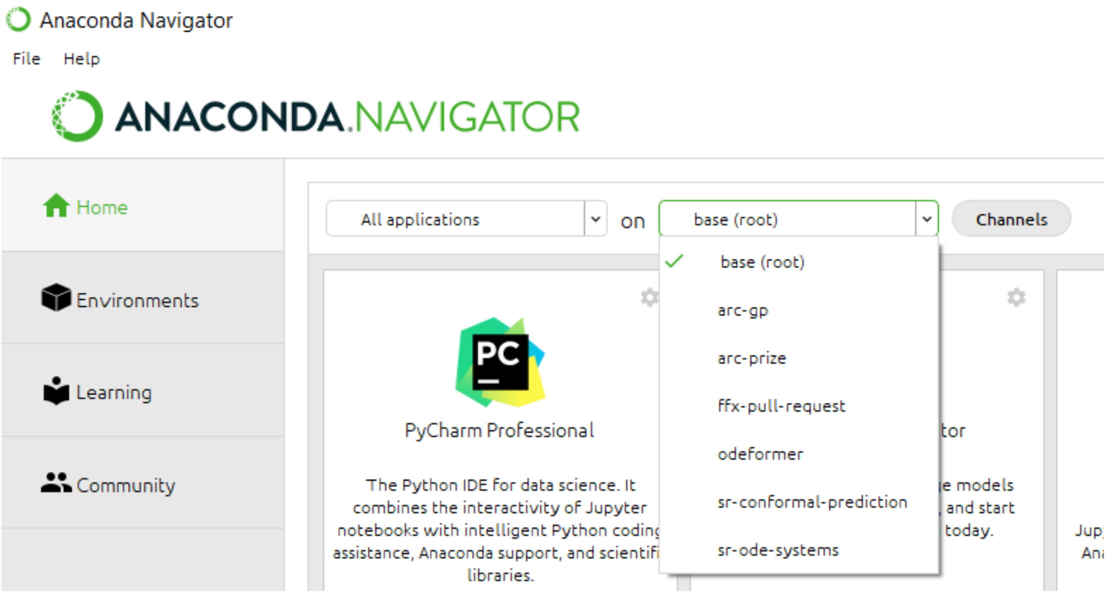
- In theory, you can edit code using a text editor
  - In practice, much better use an IDE
  - Editor, debugger, help installing packages, autocompletion, etc.
  - Some IDEs also have AI code autocompletion! *Don’t trust it 100%*
- For Python, I have experience with
  - Anaconda suite and Spyder (IDE)
  - Visual Studio Code
- PyCharm is another popular IDE

Integrated Development Environment

- Let’s install the full Anaconda suite!
- If you already have Anaconda, install VSCode

Environments

- Set of installed libraries
  - There are several ways of obtaining **separate environments**
  - Why should we do that?
  - Install only needed packages, and **specific versions**
  - Create minimal list of packages
  - Obtain **requirements.txt**



Coding: basics

- What happens when you click “Run” on your IDE?
  - “python name_of_your_script.py” in the background
  - The directory where your script is run is the **working directory**
  - It’s important to know, for relative paths, e.g. “../data/myfile.txt”
  - For example, Spyder and VSCode used different choices of **wd**
  - In doubt, check working directory with **pwd()**
  - You can change the working directory in the code, if needed
- In general, get used to Google around for information
- Or ask ChatGPT/Copilot (but beware!)

Coding: variables and data types

- Python has several **basic data types**
  - Booleans (**bool**), ints (**int**), floating point (**float**), strings (**str**), tuple
  - Lists (**list**), dictionaries (**dict**)
  - Other data types are usually **objects** imported from packages
- Why do data types exist?
- Python is an untyped (or semi-typed) language
  - As in, you are not forced to declare the type of each variable
  - Python interpreter “guesses” data type from “a = 12” or “b=2.5”
  - Typing your code can be useful for debugging or readability

Coding: variables and data types

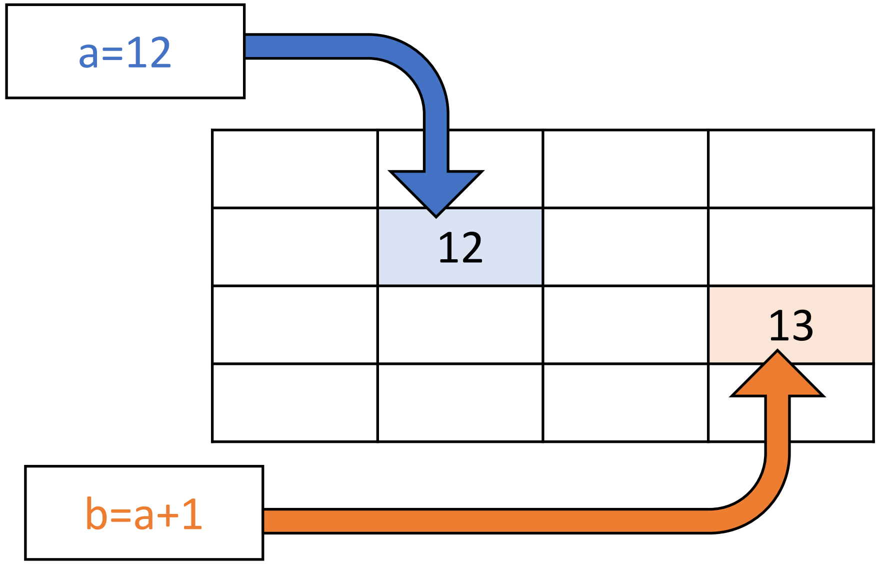
- In Python, you can declare variables anywhere in the code
  - Variables are only accessible inside the **scope** they are declared in
  - What is a "scope"? A region (part) of a program
  - A **function**, a **loop***, a **branch***
  - You can declare **global variables** (accessible from everywhere)…
  - …but don't do it; just never do it

- https://github.com/albertotonda/crash-course-data-science/

Value and reference

- Variables can be used by **value** or by **reference**
- Assignment/use by **value**
  - The variable represents an area of memory containing a **value**
  - Their **value** is copied to another area of memory (independent)
  - This is what happens for bool, int, float, string, tuple
- Assignment/use by **reference**
  - The variable represents a **pointer** to an area of memory with **value**
  - The **pointer** is copied, but still points to the same memory area
  - Modifying the copy also modifies the original (!!!)

Value and reference

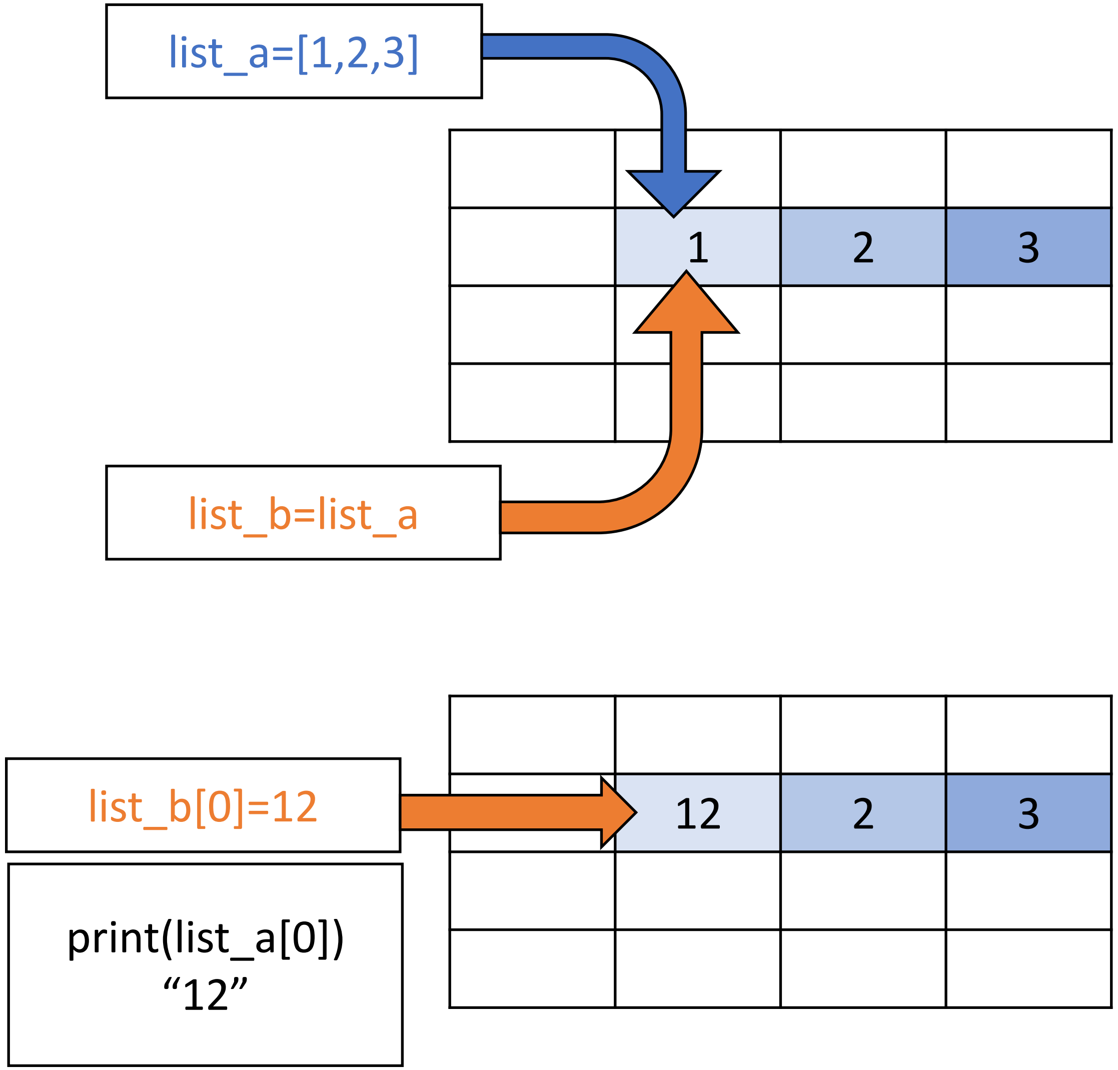


a=12

12

13

b=a+1



list_a=[1,2,3]

1 2 3

list_b=list_a

list_b[0]=12

12 2 3

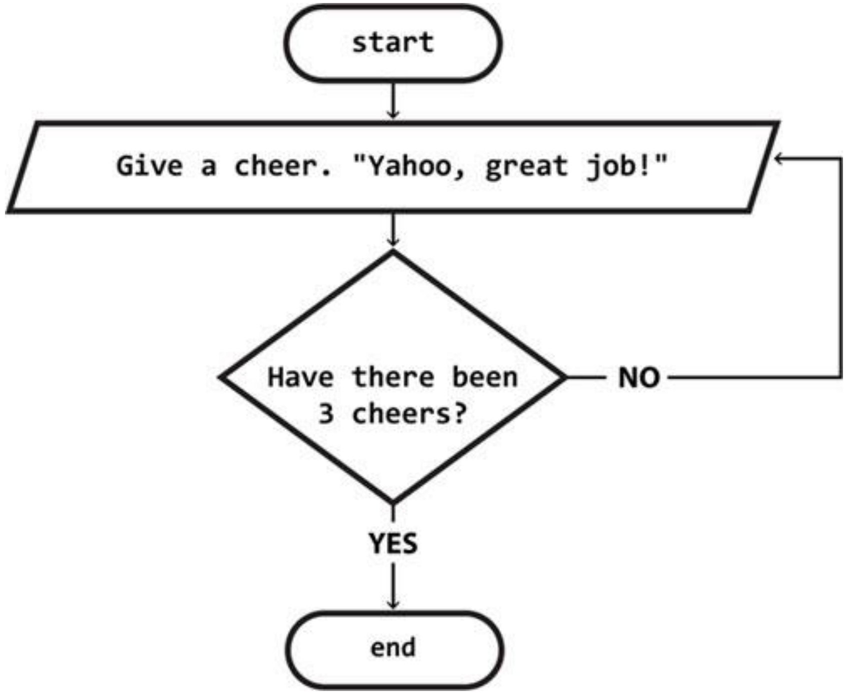
print(list_a[0])
“12”

Value and reference

- What I just said is not completely correct
  - Actually, in Python everything is stored by **reference**
  - However bool, int, float, string, tuple are **immutable**
  - So, if you try to modify an immutable, you create a copy
  - In practice, it works as assignment by value

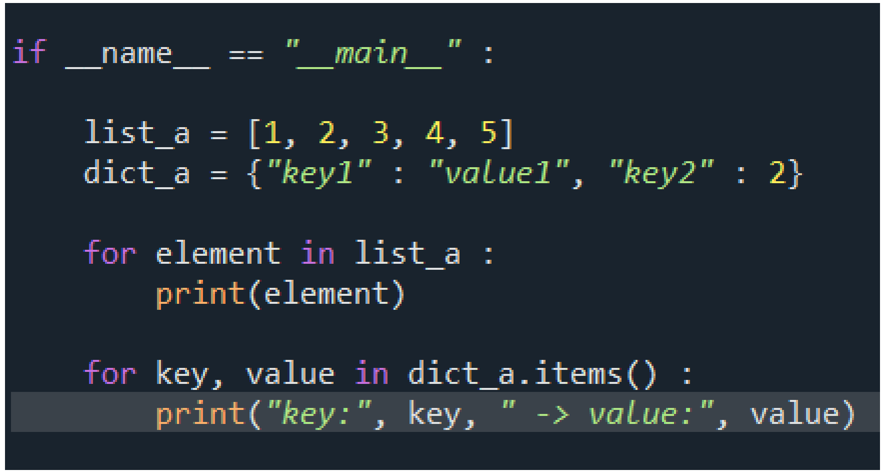
Coding: flow

- A program is typically executed top-down, line by line
- However, you can change the flow
  - Flow control (if – then – else)
  - Loops (while / for)
- Flow control is based on **Booleans**
  - True/False conditions
  - Boolean algebra! AND/OR
    - a AND b is True only if **both** a == b == True
    - a OR B is True if *either* a == True, or b == True
    - NOT a is True is a == False



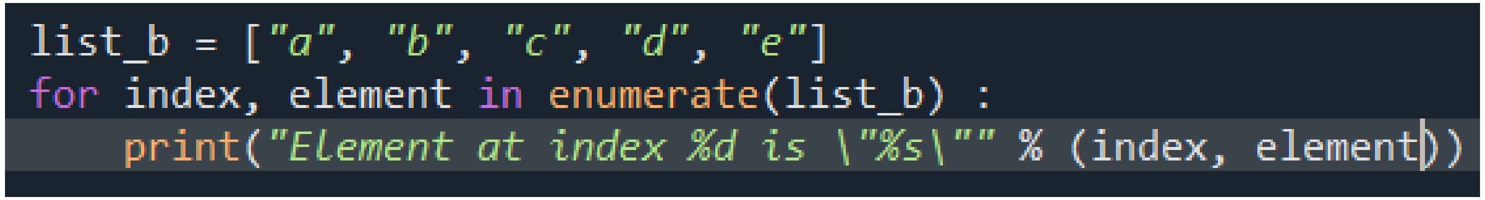
Coding: loops

- Python easily iterates over elements in list or dict



```python
if __name__ == "__main__" :

    list_a = [1, 2, 3, 4, 5]
    dict_a = {"key1" : "value1", "key2" : 2}

    for element in list_a :
        print(element)

    for key, value in dict_a.items() :
        print("key:", key, " -> value:", value)
```

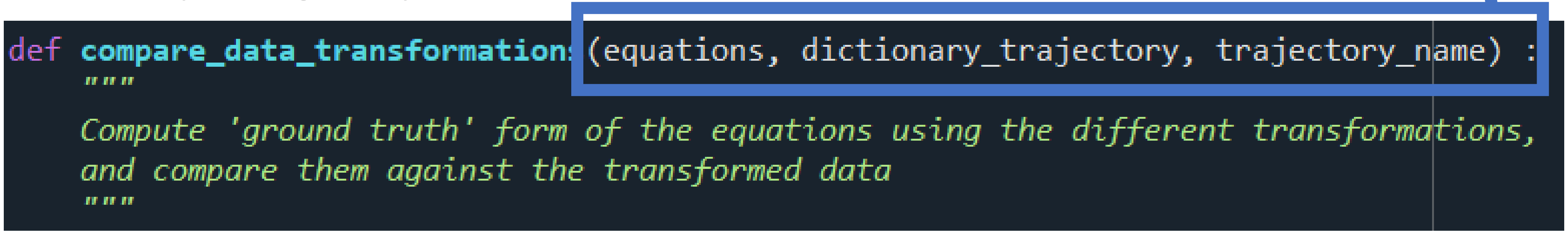
- Sometimes you need a numerical index; use **enumerate**



```python
list_b = ["a", "b", "c", "d", "e"]
for index, element in enumerate(list_b) :
    print("Element at index %d is \"%s\"" % (index, element))
```

Coding: organizing your code

- Organize your code
  - Instead of cut/pasting your code
  - Define functions, structures, objects
- Functions
  - Isolated parts (scopes) of code
  - Useful when called multiple times, or to reason in isolation
  - Everything stays inside, but beware of **references**!

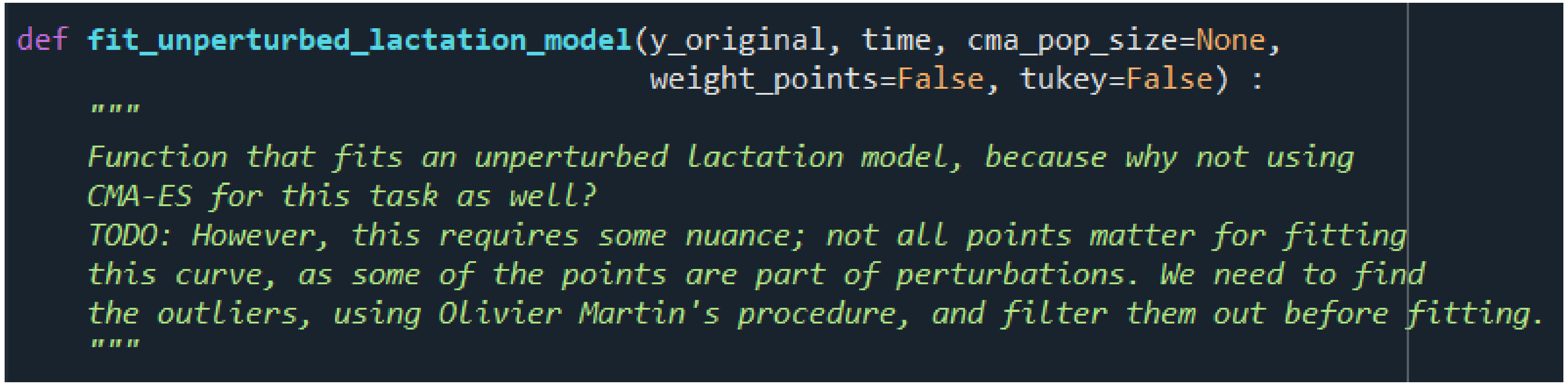


```
def compare_data_transformations(equations, dictionary_trajectory, trajectory_name) :
    """
    Compute 'ground truth' form of the equations using the different transformations,
    and compare them against the transformed data
    """
```

Signature

Coding: organizing your code

- Functions can have **default values** for arguments in signature

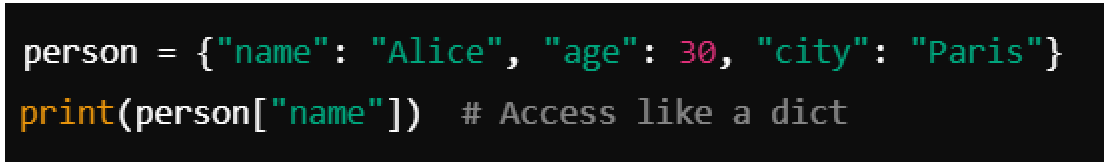


```
def fit_unperturbed_lactation_model(y_original, time, cma_pop_size=None,
                                    weight_points=False, tukey=False) :
    """
    Function that fits an unperturbed lactation model, because why not using
    CMA-ES for this task as well?
    TODO: However, this requires some nuance; not all points matter for fitting
    this curve, as some of the points are part of perturbations. We need to find
    the outliers, using Olivier Martin's procedure, and filter them out before fitting.
    """
```

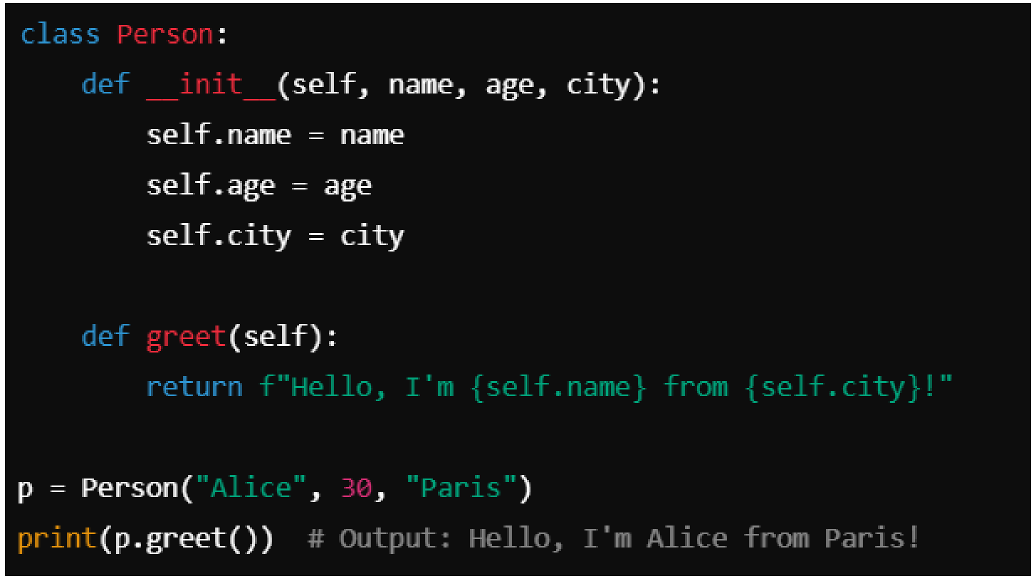
- When the function is invoked, unspecified arguments will have their default values

Coding: organizing your code

- Structures are collections of variables
  - Useful when you have information related to the same thing
  - E.g. a person’s name, surname, age, …
- In Python it’s probably better to use dictionaries or Classes



```python
person = {"name": "Alice", "age": 30, "city": "Paris"}
print(person["name"])  # Access like a dict
```



```python
class Person:
    def __init__(self, name, age, city):
        self.name = name
        self.age = age
        self.city = city

    def greet(self):
        return f"Hello, I'm {self.name} from {self.city}!"

p = Person("Alice", 30, "Paris")
print(p.greet())  # Output: Hello, I'm Alice from Paris!
```

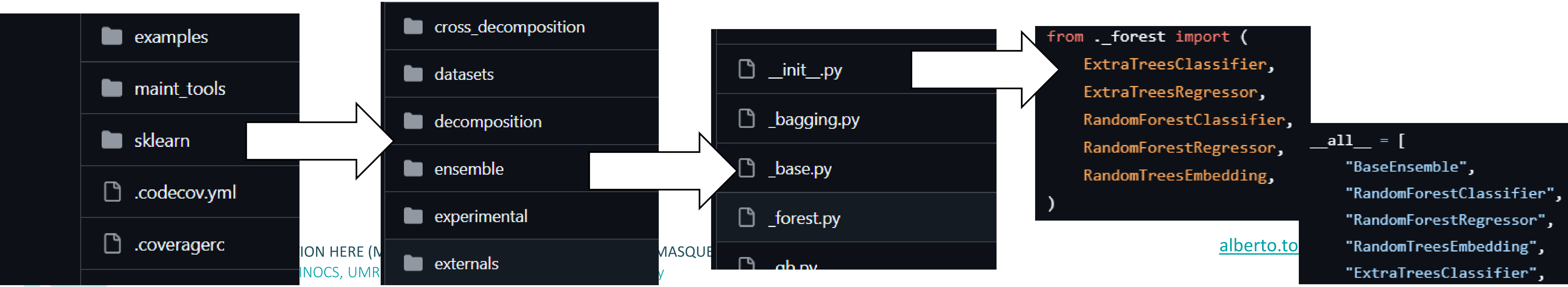
Coding: using different files

- Once you have a function, you can re-use it over and over
  - You can even put it in a separate file and **import** it!
  - If it's in the same folder, just “import <filename>”
  - Use function as “filename.my_function(argument1, argument2)”
  - Usefulness of **if __name__ == “__main__”** for imports

- If you have a lot of related, useful functions, you could even consider creating a **package** and make it available!

Coding: using different files

- In fact, you can import a whole **folder**!
  - All scripts inside the folder are accessible
  - import folder.scriptname
  - from folder.scriptname import functioname
  - Packages are structured folders, potentially with subfolders
- E.g. **sklearn.ensemble.RandomForestRegressor**



examples
maint_tools
sklearn
.codecov.yml
.coveragerc

cross_decomposition
datasets
decomposition
ensemble
experimental
externals

__init__.py
_bagging.py
_base.py
_forest.py
_gb.py

```
from ._forest import (
    ExtraTreesClassifier,
    ExtraTreesRegressor,
    RandomForestClassifier,
    RandomForestRegressor,
    RandomTreesEmbedding,
)
```

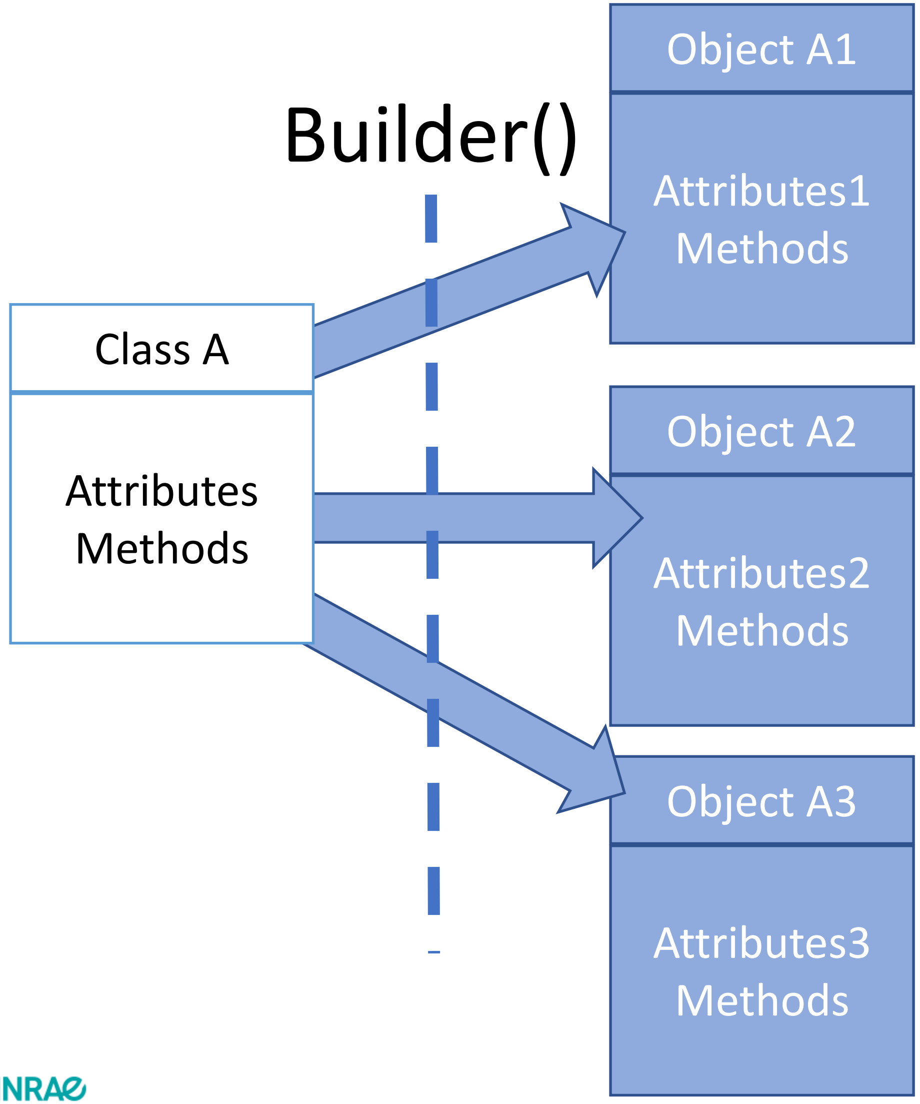
```
__all__ = [
    "BaseEnsemble",
    "RandomForestClassifier",
    "RandomForestRegressor",
    "RandomTreesEmbedding",
    "ExtraTreesClassifier",
```

Coding: organizing your code

- Different views
  - Classic: structures and functions
  - Object-oriented: classes, attributes, and methods

- Objects are instances of the same Class
  - Each object can contain variables, called **attributes**
  - And functions, called **methods**
  - Why add functions to an object? Maybe you need to perform some procedure that depends on its attributes

Coding: objects



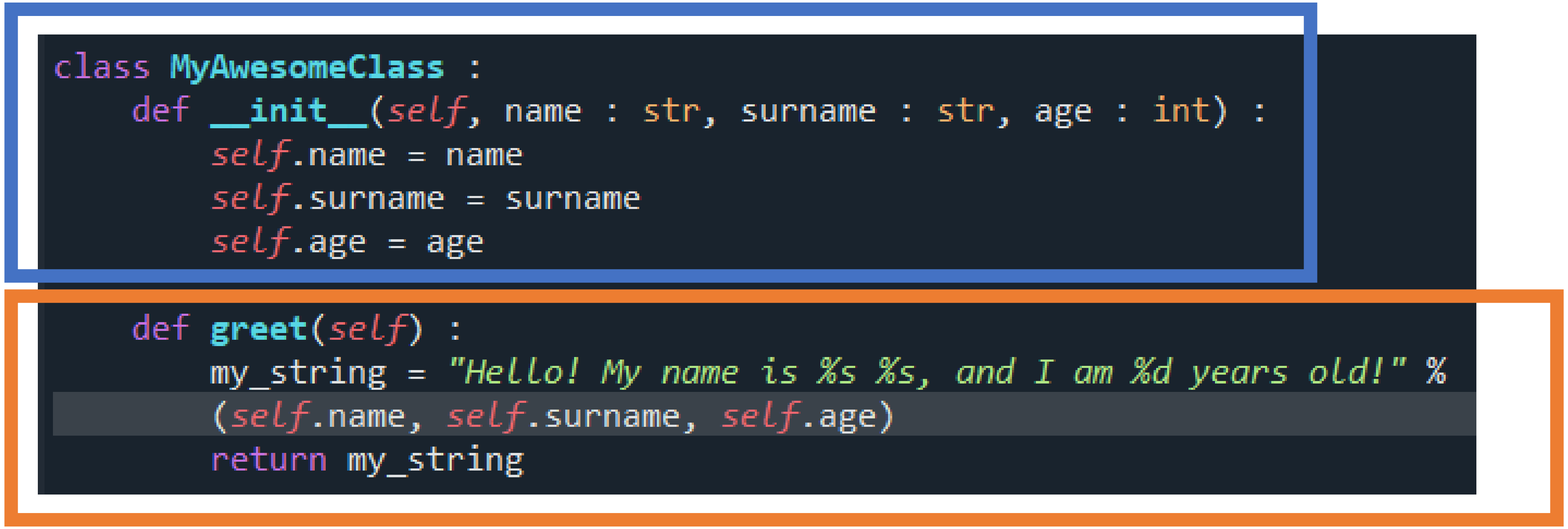
Objects A1…A3 are also called **instances** of Class A

Python syntax

- Attributes: object.attribute
- Methods: object.method()

Coding: objects

- Builder method (always called *__init__()*)
  - Assigns values from the arguments to the internal attributes
  - *self* is a reference to the current instance of the Class



Builder

```
class MyAwesomeClass :
    def __init__(self, name : str, surname : str, age : int) :
        self.name = name
        self.surname = surname
        self.age = age
```

Other method

```
    def greet(self) :
        my_string = "Hello! My name is %s %s, and I am %d years old!" %
        (self.name, self.surname, self.age)
        return my_string
```

Coding: objects

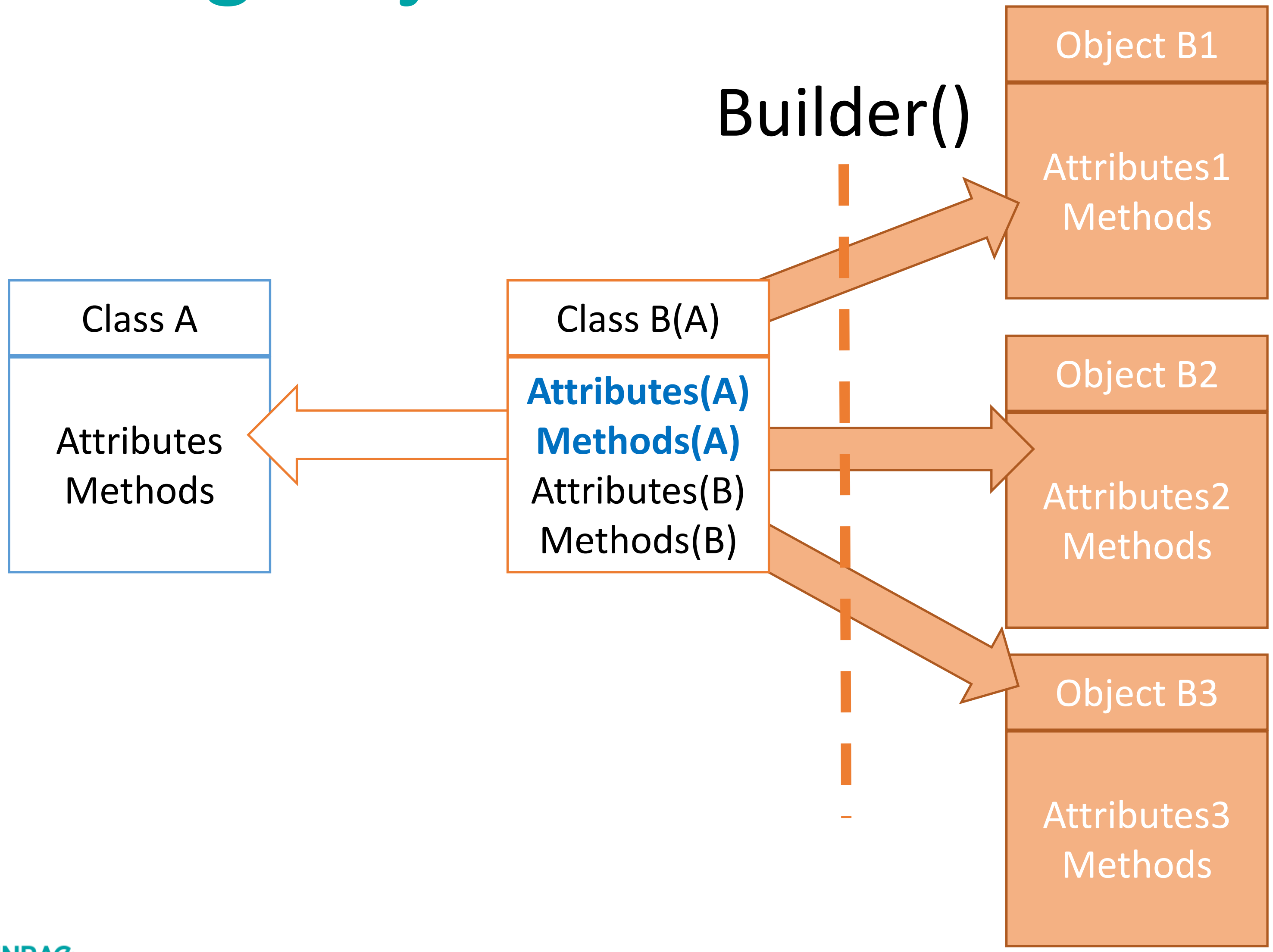
- An interesting method: __str()__
  - Generates a string representation of your objects
  - Called automatically if someone tries to **print(object)**

- Can we implement it for MyAwesomeClass?

Coding: objects

- Object-oriented programming is a dominant paradigm
- Why? Several advantages
  - Using objects is easier* than structures and functions
  - Classes can **inherit** attributes and methods from each other
  - This allows for easy creation of new classes with same methods

Coding: objects

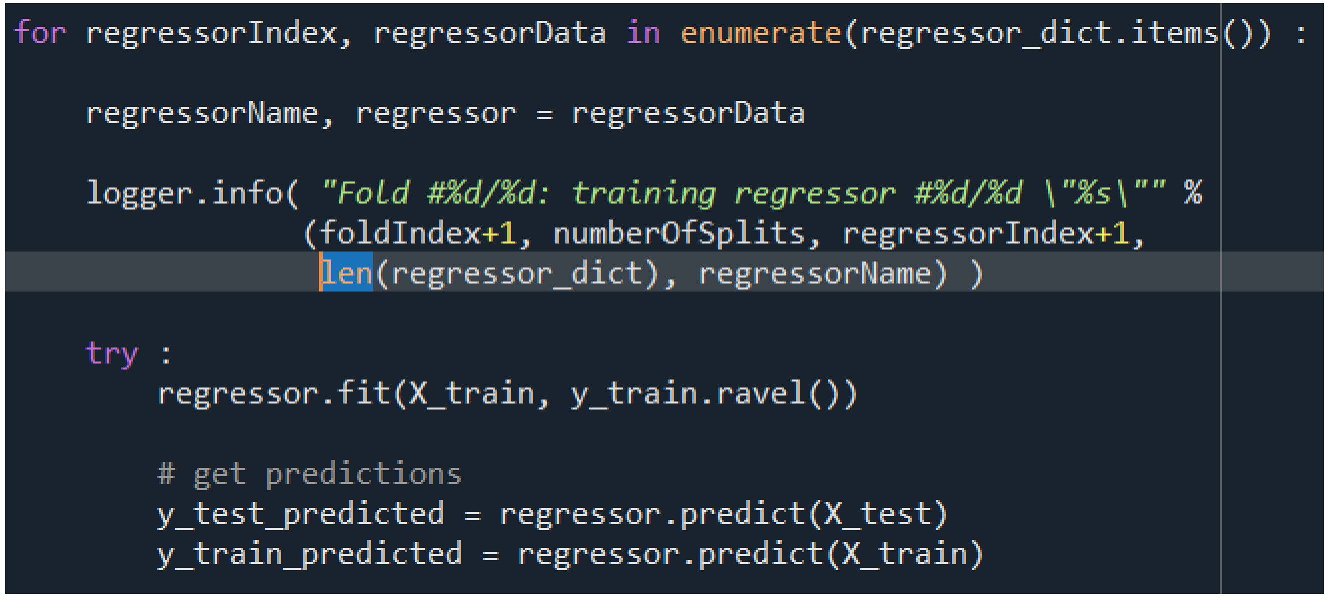


Objects B1…B3 are **instances** of Class B

Class B **inherits** (some) attributes and methods from Class A

Example: scikit-learn

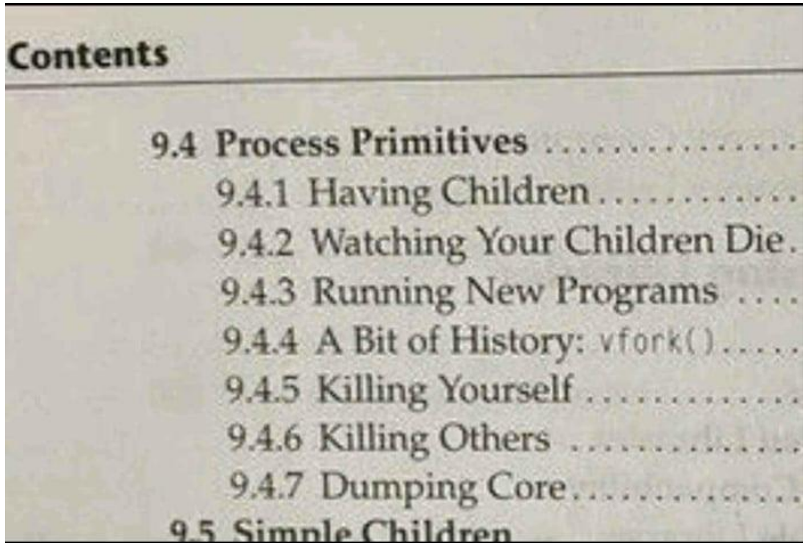
- Regressors and classifier inherit from **BaseEstimator**
  - So they all share the same methods **.fit()** and **.predict()**
  - We can use all of them with the same calls!
  - Even if we don’t know the exact type of regressor!



```
for regressorIndex, regressorData in enumerate(regressor_dict.items()) :

    regressorName, regressor = regressorData

    logger.info( "Fold #%d/%d: training regressor #%d/%d \"%s\"" %
                (foldIndex+1, numberOfSplits, regressorIndex+1,
                 len(regressor_dict), regressorName) )

    try :
        regressor.fit(X_train, y_train.ravel())

        # get predictions
        y_test_predicted = regressor.predict(X_test)
        y_train_predicted = regressor.predict(X_train)
```

Coding: parallel computations

- Modern microprocessors can run several programs in parallel
  - Multi-threading or multi-processing
  - This can save a lot of time! But it’s not easy, and OS-dependent
- Multi-process
  - Copies the whole memory of the program
  - It’s a separate scope, it needs to **return** info
- Multi-thread
  - All parallel processes share same memory
  - They need to coordinate to access it!

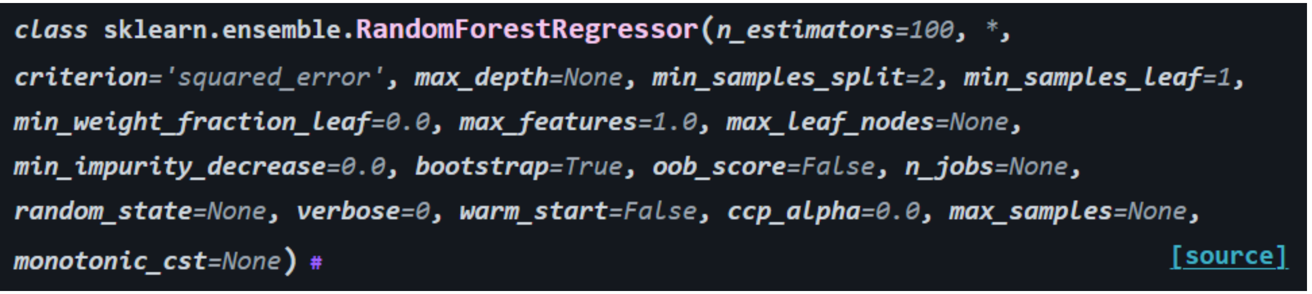


Contents

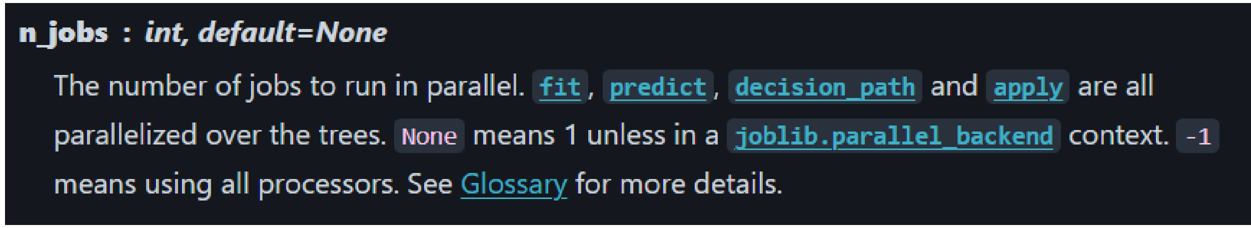
9.4 Process Primitives ....
9.4.1 Having Children ....
9.4.2 Watching Your Children Die .
9.4.3 Running New Programs ....
9.4.4 A Bit of History: vfork() .....
9.4.5 Killing Yourself ....
9.4.6 Killing Others ....
9.4.7 Dumping Core.............
9.5 Simple Children

Coding: parallel computations

- Python is often not super-efficient with multithreading
  - This is due to the Global Interpreter Lock
  - Go for multiprocessing, if you can
- In many cases, parallelization has been done for you
- For example, scikit-learn methods have a **n_jobs** argument
- They exploit parallelism on n_jobs processes



```
class sklearn.ensemble.RandomForestRegressor(n_estimators=100, *,
criterion='squared_error', max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=1.0, max_leaf_nodes=None,
min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None,
random_state=None, verbose=0, warm_start=False, ccp_alpha=0.0, max_samples=None,
monotonic_cst=None) #                                                    [source]
```



n_jobs : *int, default=None*

The number of jobs to run in parallel. `fit`, `predict`, `decision_path` and `apply` are all parallelized over the trees. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

Coding: scripts vs notebooks

- Notebooks are good for teaching, data analysis, visualization
  - Mix of text and Python code cells; you can write notes!
  - Run everything online in **Google Colaboratory**
- However, be really *really* careful with code execution
  - Variables of previously executed cells stay in memory
  - Running same cell multiple times may lead to undesired effects
- I use notebooks for prototyping and plotting images

Make your code understandable

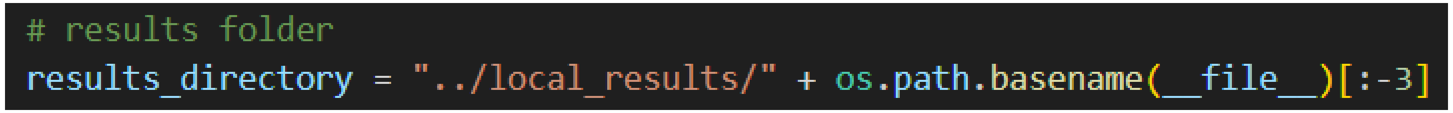
- Put A LOT OF COMMENTS EVERYWHERE
  - “Code is there to explain comments to the computer”
  - Your future self will thank you!

- Naming conventions
  - `more_readable_with_underscores` than `CamelCase`
  - `everything_lowercase`, `ClassNames` are uppercase/camel

Make your code understandable

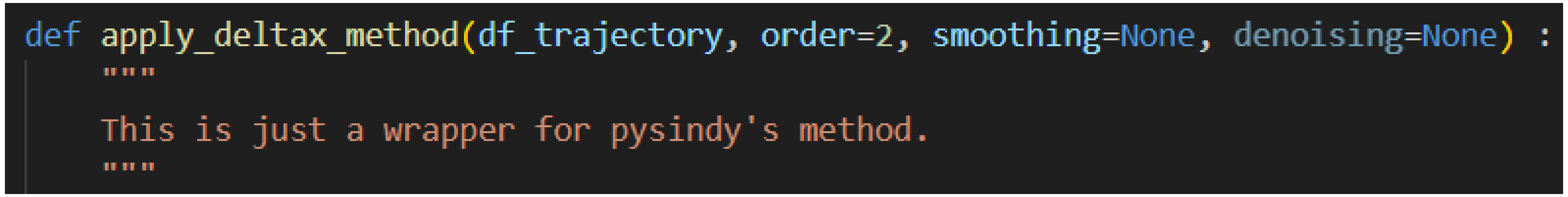
- Coders used to use short names for variables (memory!)

- Variable naming



```
# results folder
results_directory = "../local_results/" + os.path.basename(__file__)[:-3]
```

  - Use long names, who cares?
  - Names of variables should be just nouns

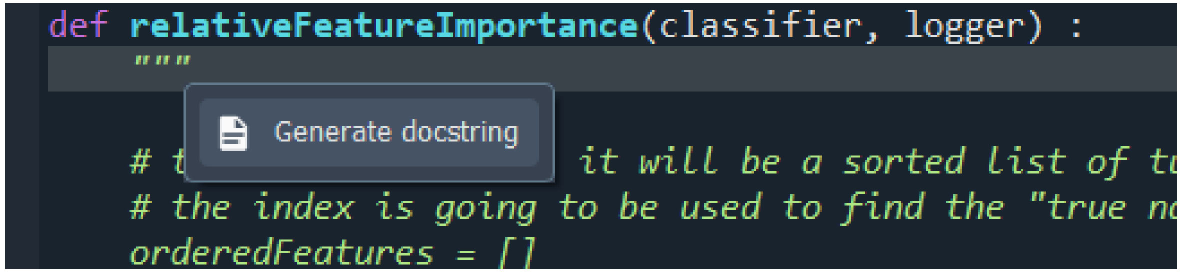


```
def apply_deltax_method(df_trajectory, order=2, smoothing=None, denoising=None) :
    """
    This is just a wrapper for pysindy's method.
    """
```

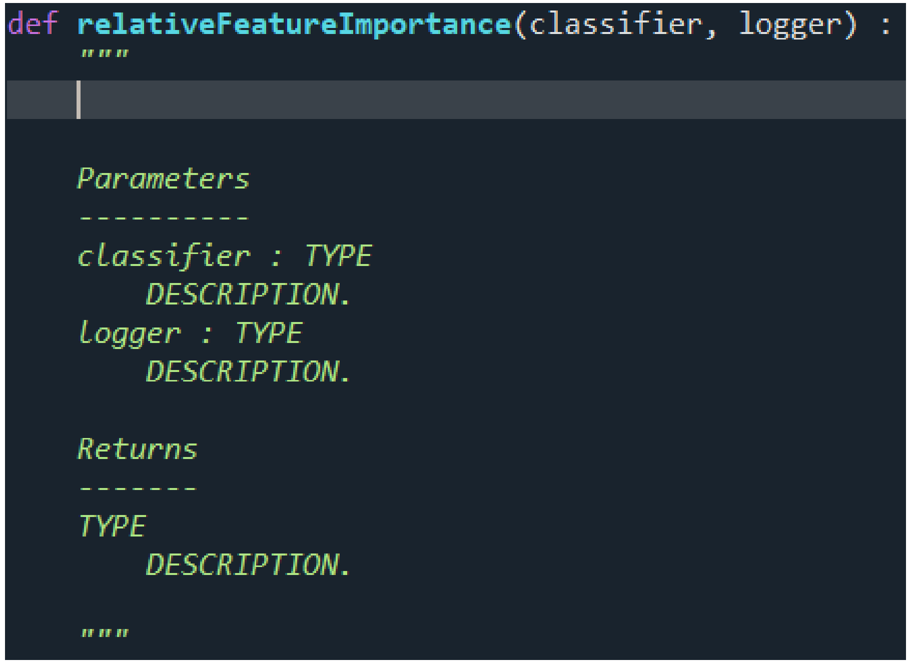
- Function naming
  - Function names should be verbs (they do something)
  - Again, you can and should use long names

Make your code understandable

- IDEs can create special comments for functions
- Can be **automatically extracted** to create **documentation**
- More work now, for less work later…?



```
def relativeFeatureImportance(classifier, logger) :
    """

    # t                           it will be a sorted list of tu
    # the index is going to be used to find the "true no
    orderedFeatures = []
```



```
def relativeFeatureImportance(classifier, logger) :
    """
    |

    Parameters
    ----------
    classifier : TYPE
        DESCRIPTION.
    logger : TYPE
        DESCRIPTION.

    Returns
    -------
    TYPE
        DESCRIPTION.

    """
```

Make your code understandable

- Example
  - https://github.com/albertotonda/HumanModels
  - https://humanmodels.readthedocs.io/en/latest/

Refactoring

- Once a first version of your code runs, **reorganize it**
  - Start from an empty repository, and create new files
  - Long, boring, and high risk of introducing bugs
  - Nobody wants to do it, but it’s extremely useful long-term
  - It’s a good moment to structure your own **package**

Packaging

- It’s actually easy* to create your own Python package
- Other people will be able to easily* install it

1. Organize the code in your repository in a specific way
2. Add some files, following a step-by-step guide
3. Upload your package to **pypy**
4. Install it using **pip install <package_name>**

alberto.tonda@inrae.fr