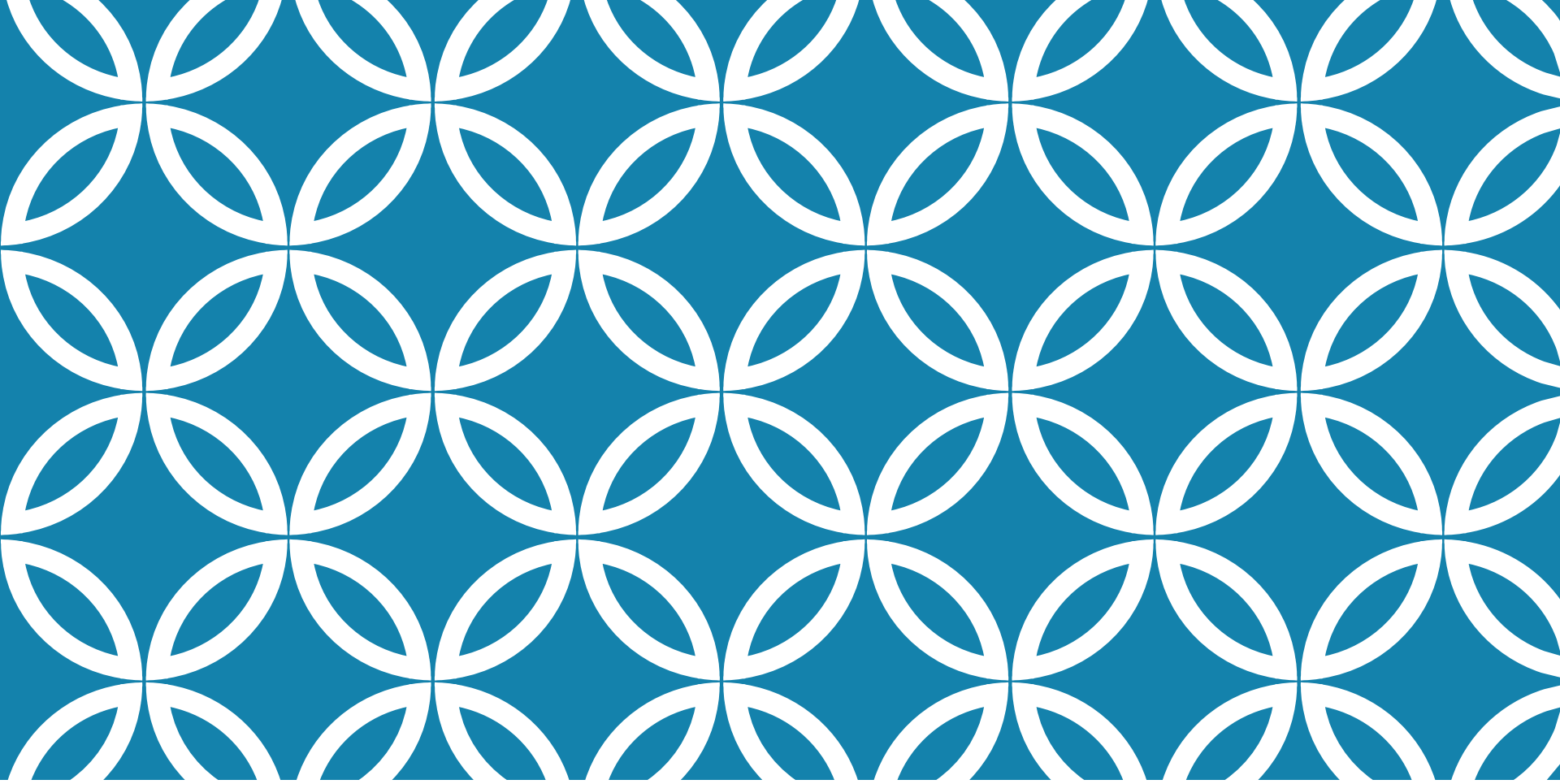


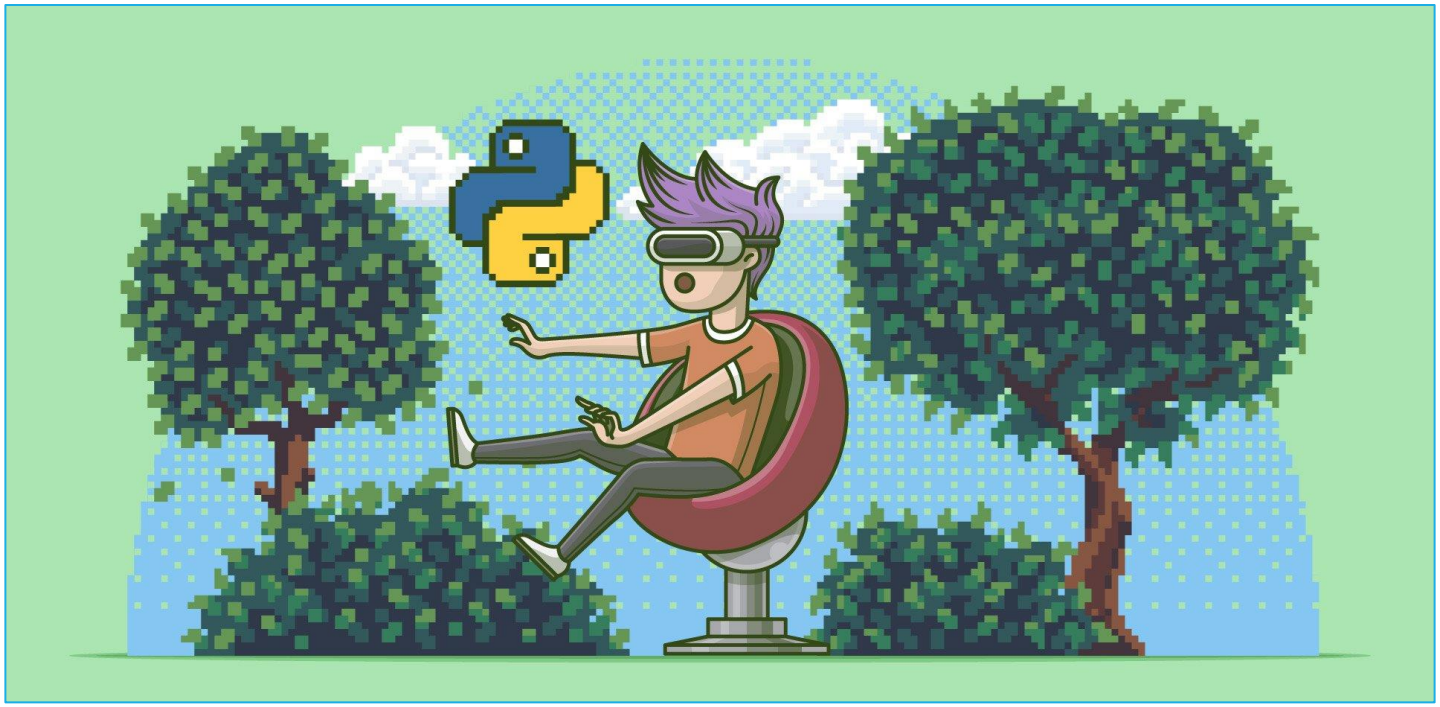
USING PYCHARM, PYLINT, AND GITHUB TO DESIGN A PROJECT PIPELINE IN PYTHON

Antonio Luca Alfeo

OVERVIEW

1. Introduction to Python and Virtual environment
   I. Package managing
   II. Anaconda
2. Pycharm Installation
3. Python basics
   I. Indentation
   II. Types and Variables
   III. Conditional Instructions
   IV. Loops
   V. Functions
   VI. Classes
4. Work with Pycharm projects
   I. Interpreter config
   II. Requirements management
   III. Project navigation and run
5. Basics to design a structured project pipeline
   I. Project organization
   II. Tabular data with pandas.dataframes
   III. Data segregation with Sklearn
   IV. Model design
   V. From JSON to Object
   VI. Model hyperarametrization
   VII. Performance evaluation
   VIII. Model deployment
   IX. REST API as interfaces
6. Code quality checking
   I. PEP8
   II. Pylint
   III. Plugin installation and usage
7. Introduction to Git and Github
   I. Repo
   II. Commit
   III. Branch
   IV. Merge
   V. Remotes
   VI. Github
8. Version Control on Pycharm
   I. Local history
   II. Connect to GitHub
   III. Github: share a project
   IV. Github: commit and push
   V. Github: clone a project

INTRODUCTION TO PYTHON

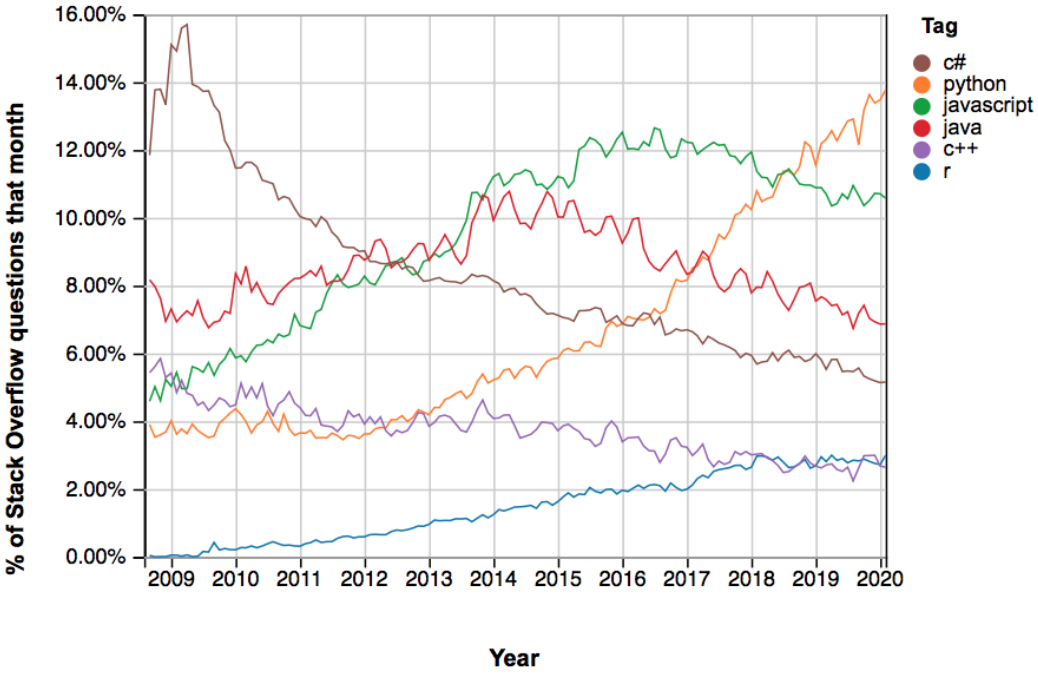


Based on previous lecture by F. Galatolo

WHY PYTHON?

Created by **Guido van Rossum** in the 1980s, to be a general-purpose language with a simple and readable syntax. Today is more and more required since it is:

- High level
- Open source
- Portable: write once run everywhere
- Extendible in C/C++
- Easy to learn
- With a mature and supportive community
- With hundreds of thousands of libraries, packages and frameworks, supported by big players (Google, Facebook, Amazon) and non-profit organizations



VIRTUAL ENVIRONMENT

- In each project, a number of Python packages are imported and used. Each of them may requires a given version of other packages and Python as well.
- A **virtual environment** is a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages
- In this way, the project-wide dependencies are stored, easily snapshotted and retrieved
- Create a Virtual Environment with Python X.Y in folder env

```
virtualenv --python=pythonX.Y env
```

- Activate the Virtual Environment

```
source ./env/bin/activate
```

```
. ./env/bin/activate
```

- Deactivate the Virtual Environment

```
deactivate
```

BASIC PACKAGES MANAGING

- Install package

```
pip install package
```

- Uninstall package

```
pip uninstall package
```

- Snapshot installed packages in *requirements.txt*

```
pip freeze > requirements.txt
```

- Install all packages snapshotted in *requirements.txt*

```
pip install -r requirements.txt
```

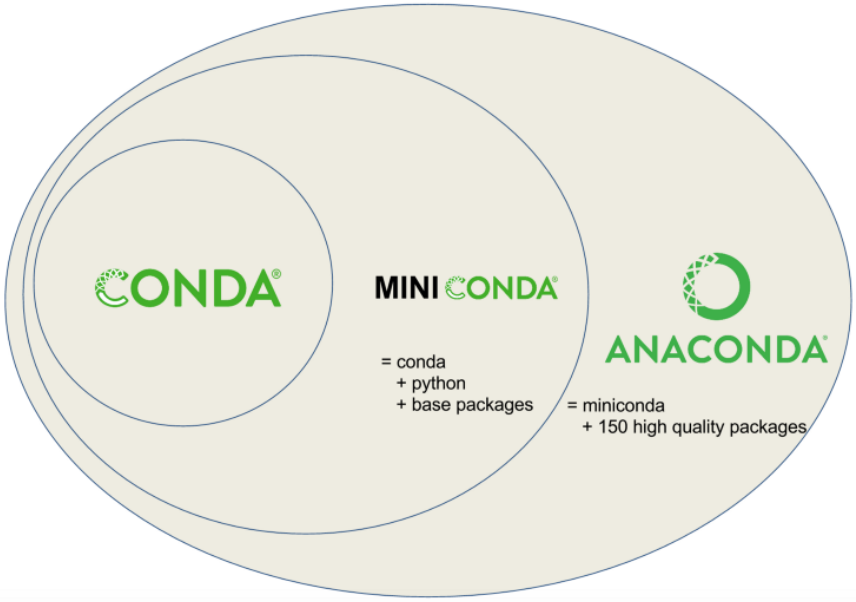
- Now you can import and use this packages in your project

ANACONDA

Anaconda is a distribution of Python that aims to simplify package management and deployment, suitable for Windows, Linux, and macOS.

Package versions in Anaconda are managed by **conda**, an open source, cross-platform, language-agnostic package manager and environment management system

Conda analyses the current environment including everything currently installed, works out how to install/run/update a compatible set of dependencies



PYTHON IDE + MINICONDA = PYCHARM

Pycharm offers **configurable python interpreter** and **virtual environment support**.

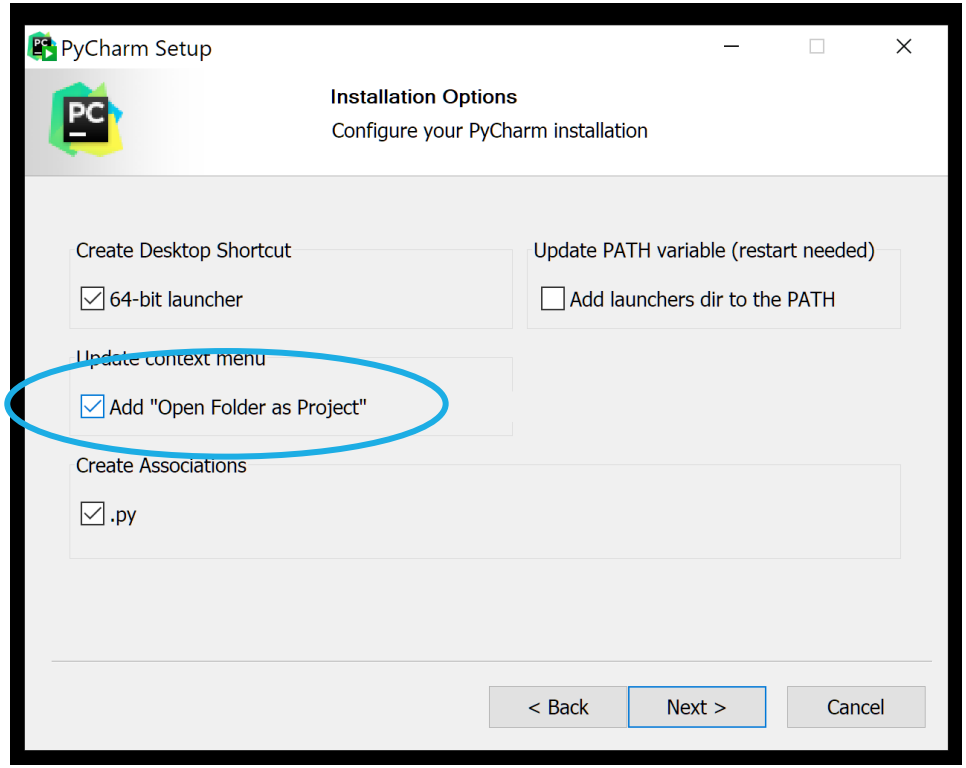
Based on previous lecture by A. L. Alfeo

INSTALLATION 1/2

1. Install **PyCharm** using this links:
   - Linux
   - Windows
   - MacOs

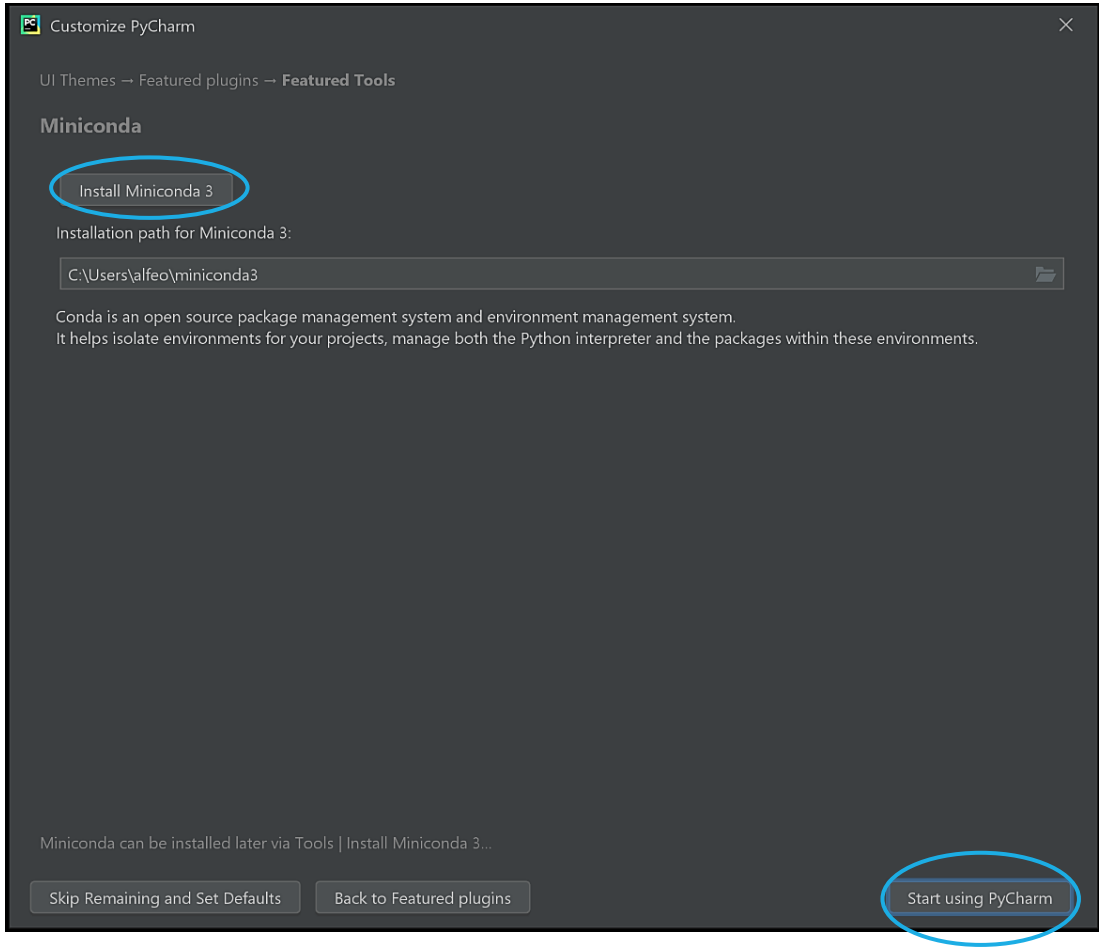
2. During PyCharm installation

   enable “open folder as project”



INSTALLATION 2/2

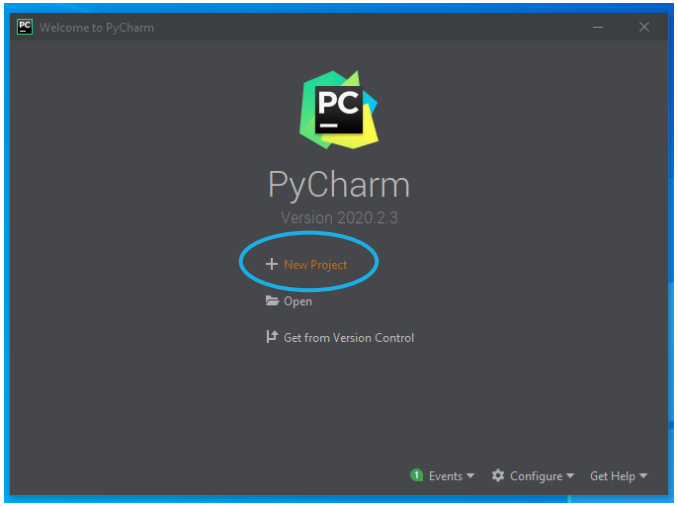
- Accept the JetBrains Privacy Policy
- Choose the UI theme you prefer
- Do not install any featured plugin
- Install Miniconda: includes the conda environment manager, Python, the packages they depend on, and a small number of other useful packages (e.g. pip).
- Remember, Miniconda can be installed at any time from **Tools -> Install Miniconda3**
- Start using PyCharm

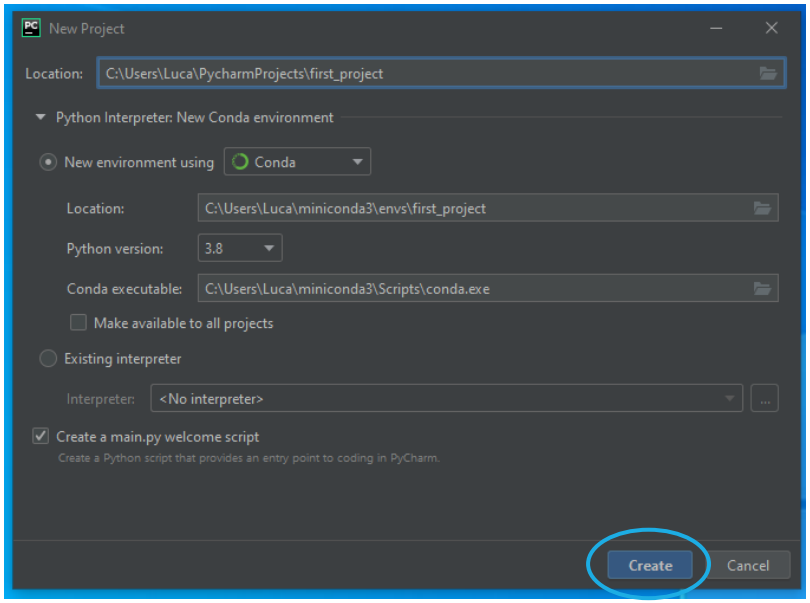


NEW PROJECT

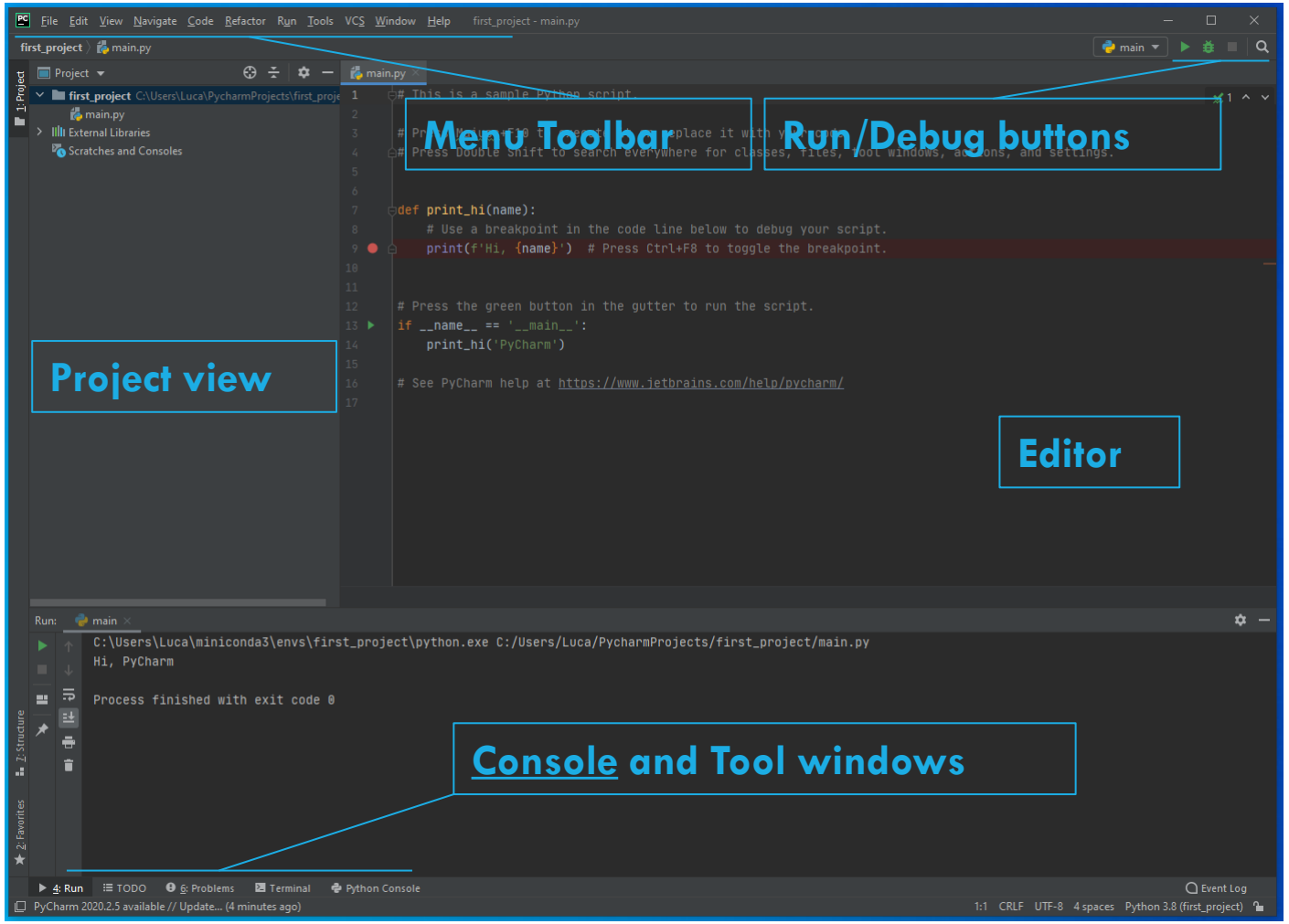
Create a new project. Name it “first_project”.

If needed set the **conda executable path**.



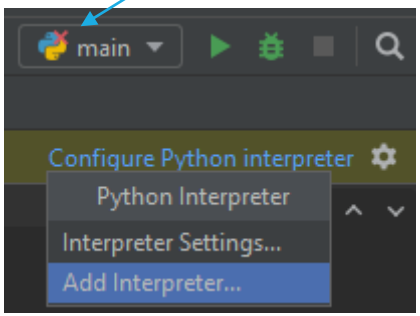


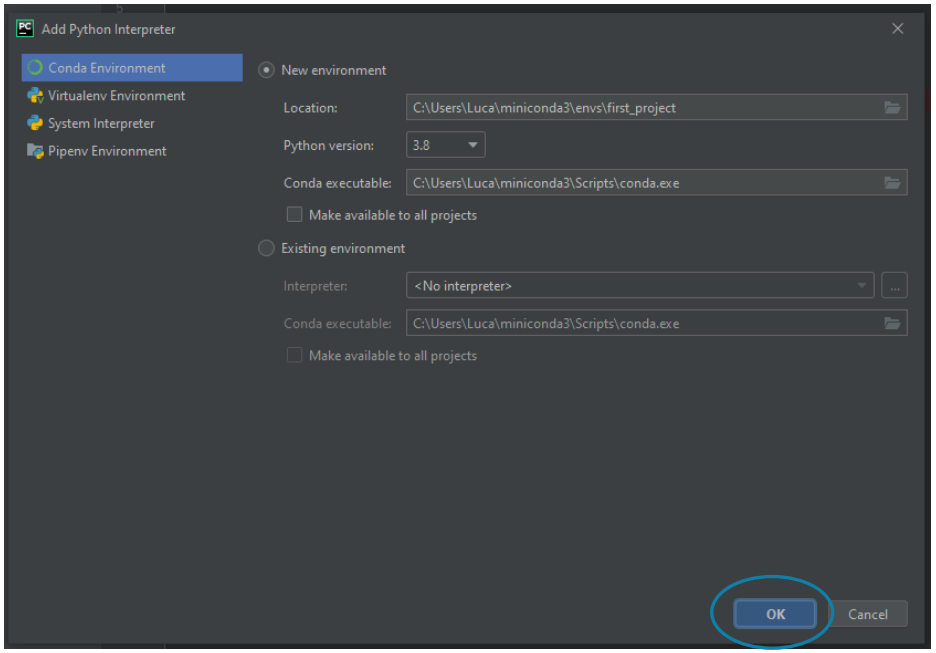
GUI


Menu Toolbar
Run/Debug buttons
Project view
Editor
Console and Tool windows

INTERPRETER: DEFAULT CONFIGURATIONS

Is there a red **X** here?





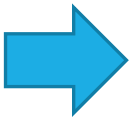
Now you can run it!

PYTHON BASICS

INDENTATION

In python **code blocks are not surrounded by curly brackets**. Just use the correct **indentation**!

```
for(int i = 0; i < n; i++){
    int k = i % 3
    if(k == 0){
        // stuff...
    }
}
```



```
for i in range(0, n):
    k = i % 3
    if k == 0:
        # stuff...
```

BUILT-IN TYPES

Python is **dynamically typed**, i.e. types are determined at runtime.

Variables can **freely change type during the execution**.

In python there are a lot of built-in types, the most notables are:

- **Boolean** (*bool*)
- **Strings** (*str*)
- **Numbers** (*int*, *float*)
- **Sequences** (*list*, *tuple*)
- **Mapping** (*dict*)

VARIABLES

Variables can be assigned to given values…

```
pi = 3
```

```
name = “Pippo"
```

…Even multiple variables at once via **iterable unpacking**. Click for more on iterable data structures (e.g. lists, tuples…)!

```
pi, name = 3, “Pippo"
```

```
first, second, third = SomeSequence
```

In Python everything is stored and passed as **reference** with the only exception of Numbers.

```
a = [1, 2, 3]
```

```
b = a
```

```
b[0] = 5 # now a = [5, 2, 3]
```

CONDITIONAL INSTRUCTION 1/2

Simple conditional instruction with the **if** keyword.

```
if someConditions:
    someActions()
    someOtherActions()
```

Python uses **and** and **or** as logical operators instead of && and ||

```
if (C1 and C2) or C3:
    someActions()
    someOtherActions()
```

Inline conditional instructions can be used as it follows

```
value if Condition else otherValue
```

CONDITIONAL INSTRUCTION 2/2

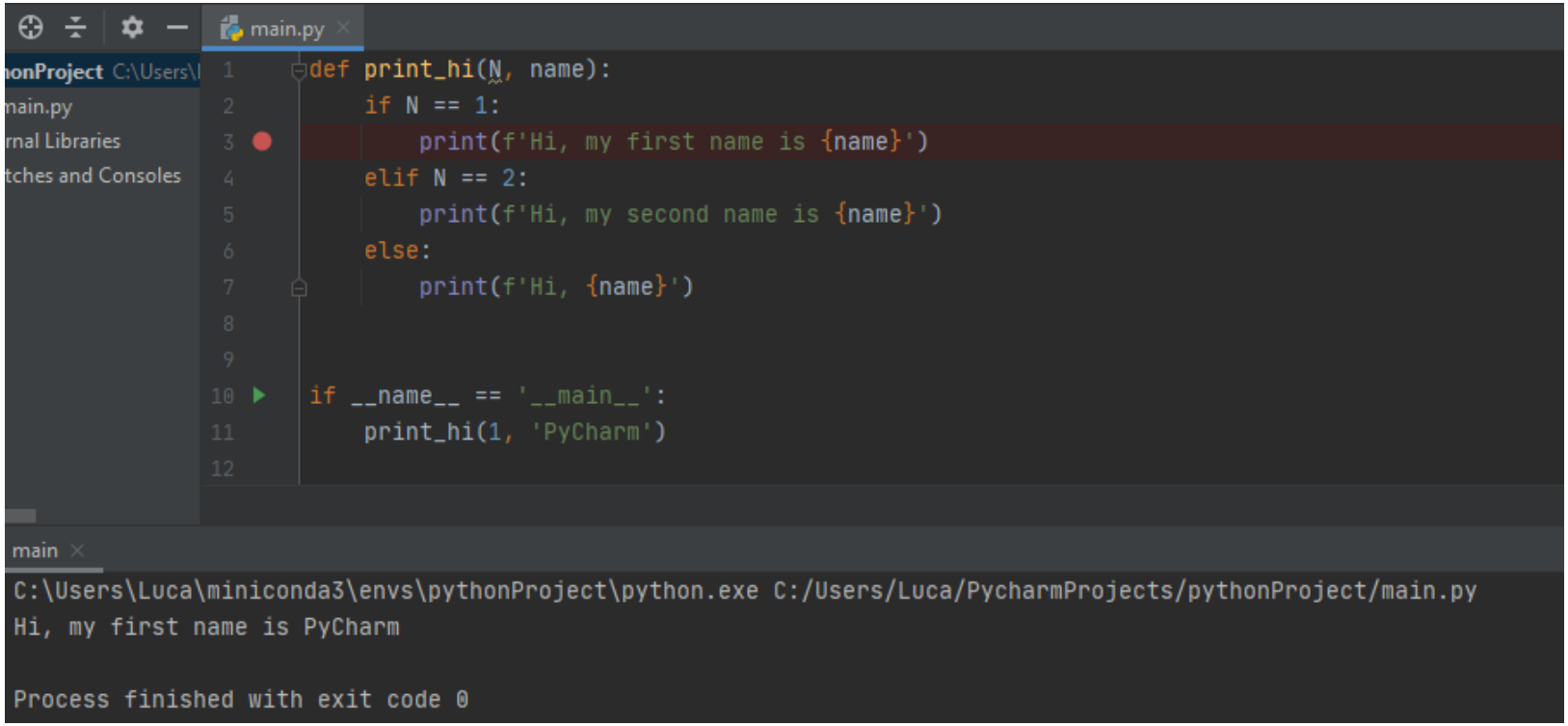
The **if-else** statement is used as it follows

```
if Condition:
    someActions()
else:
    someOtherActions()
```

There is no switch case statement in Python. Just use **if** and **elif**

```
if C1:
    A1()
elif C2:
    A2()
elif C3:
    A3()
else:
    A()
```

TRY IT YOURSELF!



LOOPS 1/2

There is no do-while loop, just **while** and **for** loops

```
while Conditions:
    Stuff()
    otherStuff()
```

```
for element in elements:
    doStuff(element)
```

Tuple unpacking can be used with loops iterations

```
for x, y in SequenceOfTuples:
    doStuff(x, y)
```

LOOPS 2/2

zip() combines one-by-one the elements of two or more iterables

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
for x, y in zip(L1, L2):
    print(x, y)
```

enumerate() returns a list of *(index, element)* tuples

```
names = ["Federico", "Mario", "Giovanni"]
for i, name in enumerate(names):
    print(i, name)
```

For efficient loops an iterable object can leverage the itertools packages.

The simplest iterable can be a **list** created via **list comprehension**

```
[someOperation(element) for element in elements]
```

```
squares = [i**2 for i in range(0, N)]
```

FUNCTIONS: DEFINITION

A new function can be defined by using the keyword **def**

```
def getCircleArea(r):
    return pi*r**2
```

Default argument are indicated with **=**

```
def getCircleArea(r, isEngineer=True):
    pi = 3 if isEngineer else 3.1415
    return pi*r**2
```

Inline functions can be created by using the **lambda** keyword

```
lambda comma, separated, arguments : expression
```

For example

```
norm2D = lambda x, y: math.sqrt(x**2 + y**2)
```

FUNCTIONS: POSITIONAL ARGUMENTS

A **variable** number of arguments can be defined via the * symbol

```
def sumOfSquares(*args):
    squares = [arg**2 for arg in args]
    return sum(squares)

result = sumOfSquares(1, 2, 3)
```

A **sequence** can be passed as positional arguments as it follows

```
def norm2D(x, y):
    return math.sqrt(x**2 + y**2)

vec = [2, 3]
norm = norm2D(*vec)
```

FUNCTIONS: KEYWORD ARGUMENTS

A function using keyword arguments needs to have the ** **symbol as last argument**.

```
def greet(language = "en", **kwargs):
    if language == "it":
        print("Ciao "+kwargs["name"]+" "+kwargs["surname"])
    else:
        print("Hello "+kwargs["name"]+" "+kwargs["surname"])

greet("it", surname="Galatolo", name="Federico")
greet(name="Mario", surname="Cimino")
```

You can also pass a **dict** of keyword arguments using the symbol ** while calling the method

```
person = dict(name="Federico", surname="Galatolo")
greet("it", **person)
greet(**person)
```

TRY IT YOURSELF!

Modify the **print_hi** function to accept a **dict** as an argument. Print:

- the first *number* squares, if *number* is greater than zero
- "Just a zero?!" if *number* is zero
- «Hi, my name is *name*», otherwise

```
def print_hi(**kwargs):
    if kwargs["number"] > 0:
        squares = [i**2 for i in range(0, kwargs["number"])]
        for elem in squares:
            print(elem)
    elif kwargs["number"] == 0:
        print("Just a zero?!")
    else:
        print("Hi, my name is " + kwargs["name"])

if __name__ == '__main__':
    argv = dict(name="Luca", number=0)
    print_hi(**argv)
```

CLASSES: METHODS

In python classes are defined with the **class** keyword. Class methods are defined with the **def** keyword. **Class method** have the first argument equal to **self**, in contrast with **static method**.

```
class Person:
    def getName(self):
        return "Federico"
    def greet(self):
        return "Hi! I am "+self.getName()
```

```
class Person:
    greeting = "Hi!"
    def getGreeting():
        return Person.greeting

g = Person.getGreeting()
```

However, unless you want to use a decorator, Python does not know about static/non-static methods, **it is all about notation!**

```
p = Person()
p.greet() # ok
Person.greet(p) # still ok
```

CLASSES: ATTRIBUTES

In python you can create, modify and retrieve **instance attributes** using the dot (.) selector on the instance reference. You can create and assign an instance attribute everywhere in a class method.

```
class Person:
    def setName(self, name):
        self.name = name
    def greet(self):
        return "Hi! I am "+self.name
```

You can create **class attributes** specifying them after the class declaration. You can modify and retrieve class attributes using the dot (.) selector on the class reference

```
class Person:
    greeting = "Hi!"
    def setName(self, name):
        self.name = name
    def greet(self):
        return Person.greeting+" I am "+self.name
```

CLASSES: VISIBILITY

In python there is no such thing as a private method or attribute. **Everything is public**. The naming *convention* for "private" methods and attributes is to precede their name with the _ symbol.

```
class Person:
    def setName(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name
```

CLASSES: CONSTRUCTOR

In python the construct function is named **__init__** and it is called at object instantiation.

You can specify one or more arguments. The first argument is the object instance reference.

```
class Person:
    def __init__(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name
p = Person("Federico")
```

CLASSES: INHERITANCE

You can **extend** a base class with another specifying the base class between the parenthesis at class definition. In order to get the base class reference you need to use the **super()** function.

```
class Person:
    def __init__(self, name):
        self._name = name
    def greet(self):
        return "Hi! I am "+self._name
```

```
class Student(Person):
    def greet(self):
        return "Leave me alone, I have to study"
```

TRY IT YOURSELF!

Include the modified version of **print_hi** in the **class** *Person*, use the class attributes, and derive the new class *Student*!

```
class Person:
    def __init__(self, name, number):
        self.name = name
        self.number = number

    def print_hi(self):
        if self.number > 0:
            squares = [i**2 for i in range(0, self.number)]
            for elem in squares:
                print(elem)
        elif self.number == 0:
            print("Just a zero?!")
        else:
            print("Hi, my name is " + self.name)
```

```
class Student(Person):
    def print_hi(self):
        print("I'm a student!")

if __name__ == '__main__':
    p = Person("Luca", 0)
    p.print_hi()
    print(p.name)

    s = Student("Luca",  -2)
    s.print_hi()
    print(s.name)
```

MORE ON CLASSES: DATA MODEL

The are a lot of built-in functions provided by the base class of all the classes object. Each of which provide a specific behavior, a few are:

- __len__ (self)

  Returns the "length" of the object (called by len())

- __str__ (self)

  Returns the object as a string (called by str())

- __lt__ (self, other), __lt__ (self, other), __eq__ (self,other)

  Called when the object is used in a comparison

- __getitem__ (self, key), __setitem__ (self, key, value)

  Called in square brackets access

WORKING WITH EXISTING PROJECTS

Pycharm provides different functionalities for project and code navigation.

Navigate the project with specialized project views, use shortcuts, jump between files, classes, methods and usages.

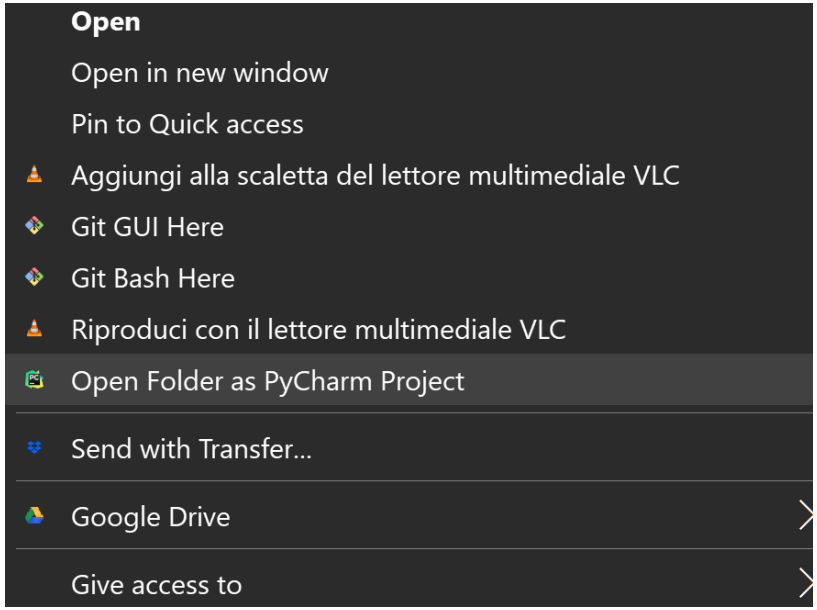
You can create a Python project by opening a folder as a project or even from scratch, by creating or importing the python files.

In this lecture we will not start from scratch…

OPEN A PROJECT

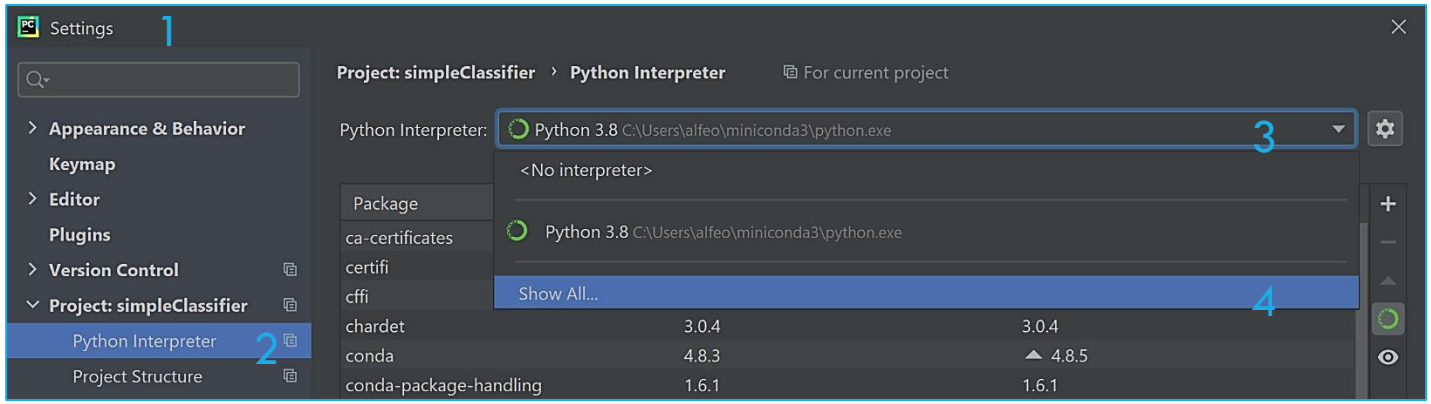
1. Unzip "simpleClassifier.rar"

2. Open the resulting folder "as a PyCharm project"



ALTERNATIVE INTERPRETER CONFIGURATION

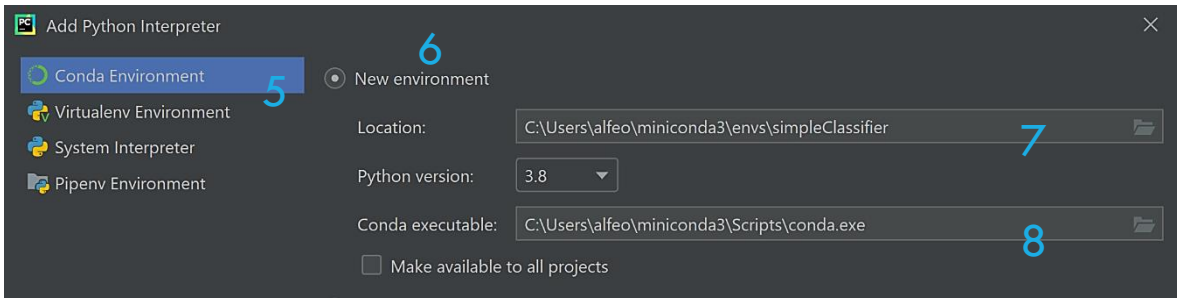
File > Settings > Project > Python interpreter > Show all



+ > Conda Environment > New environment

Location and conda executable may have slightly different paths (the first part) according to your miniconda3 location.

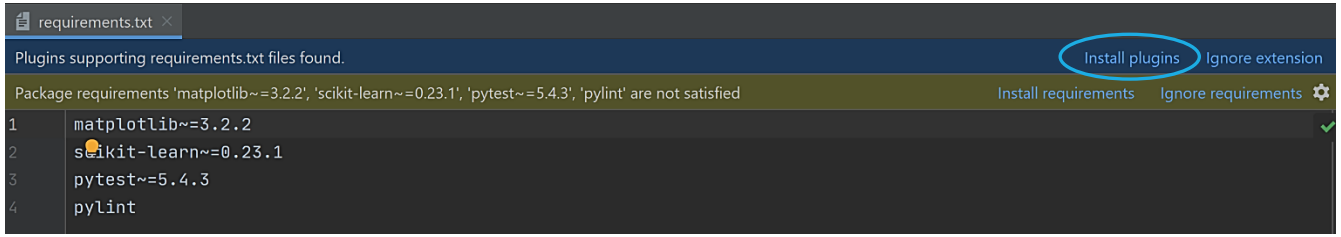
Apply > Ok



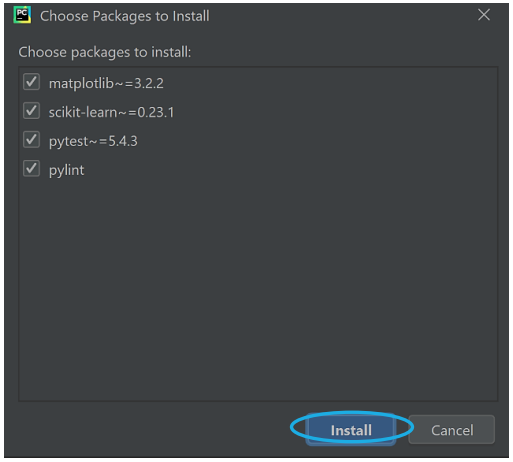
PROJECT REQUIREMENTS 1/2

The requirements are all the packages that our software needs to run properly. Those can be installed with a PyCharm plugin.

Double click on "**requirements.txt**" in the Project View > Install Plugin

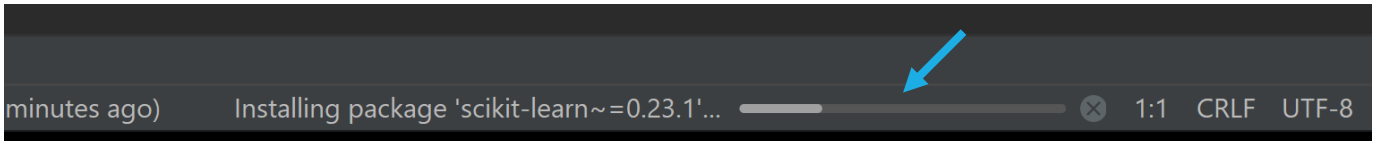


Once the plugin is installed click on "Install requirements", select all and click install.

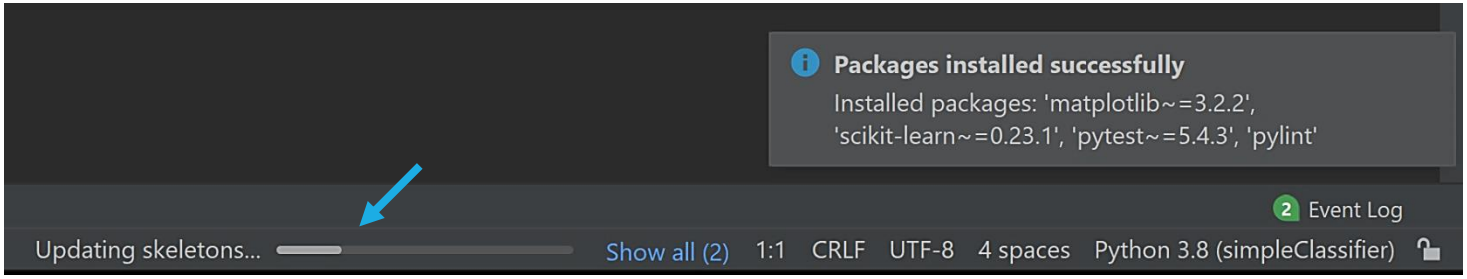


PROJECT REQUIREMENTS 2/2

- If everything goes smoothly you might see the package installation status progress at the bottom of the GUI.



- Once the package installation is done a notification appears, but it is still **NOT** possible to go to next step, at least until PyCharm has finished "Updating skeletons".



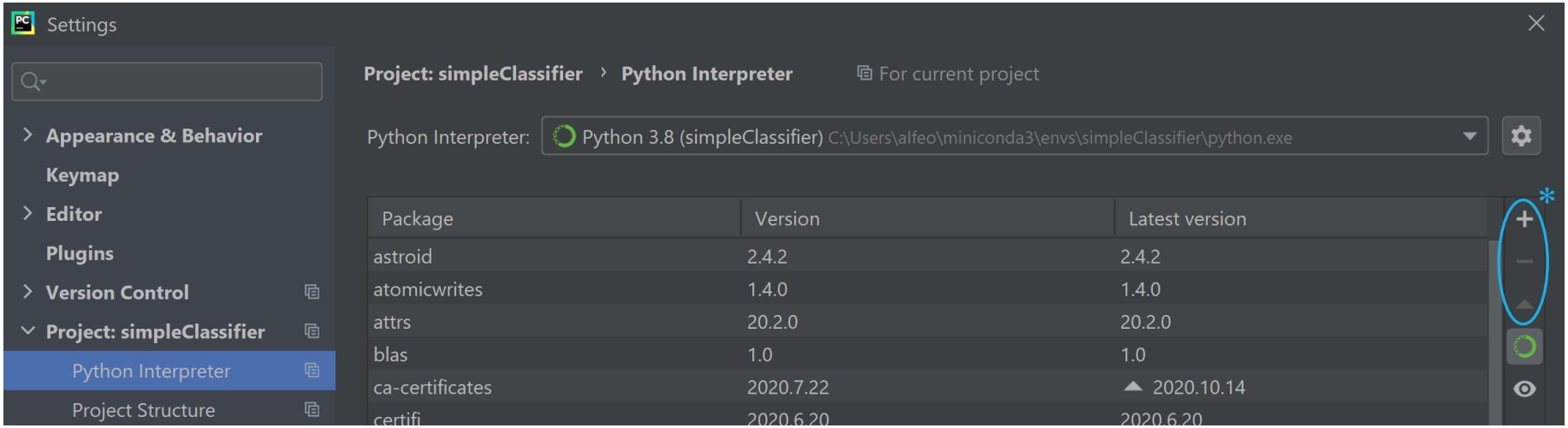
- Each package can also be installed via GUI or with the Terminal by using
  - "pip install <name_lib>" to install a single package
  - "pip install –r requirements.txt" to install the packages on the requirements.txt

CHECK THE PACKAGES AVAILABLE

It can take some minutes to complete the requirements installation. A restart may be required before moving to the next step!

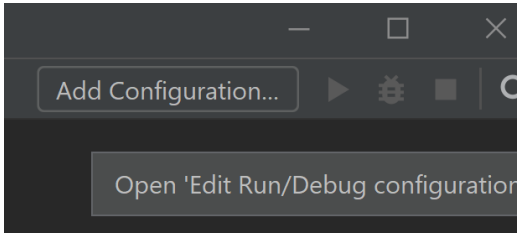
Once done you can see (add, eliminate and upgrade*) the packages and libraries available in your virtual environment by checking :

File > Settings > Project > Python Interpreter

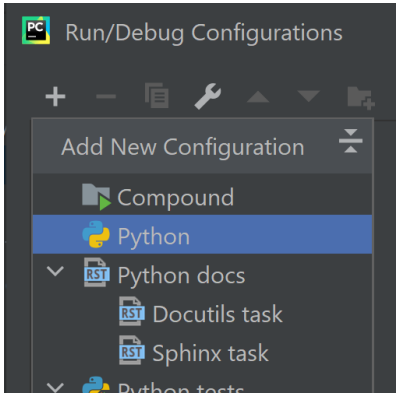


CONFIGURE THE FIRST RUN

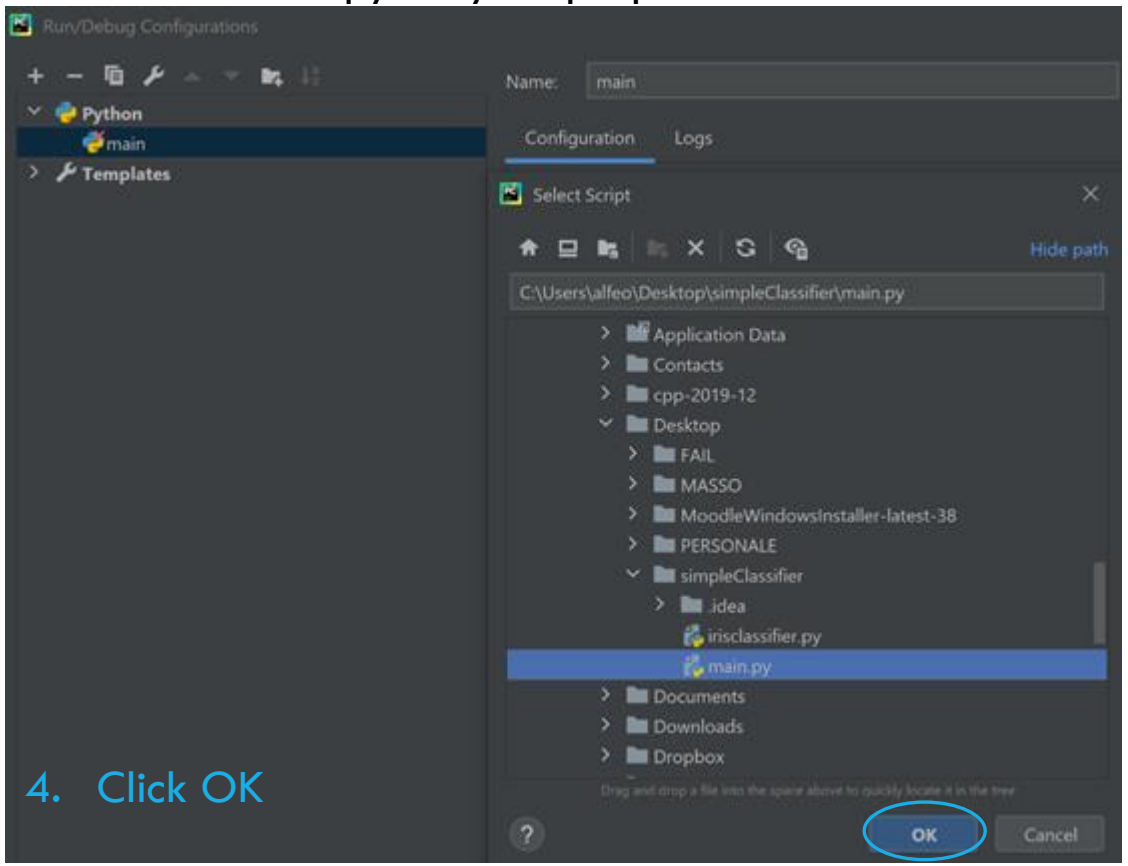
1. Click “Add configuration”



2. Select “Python”



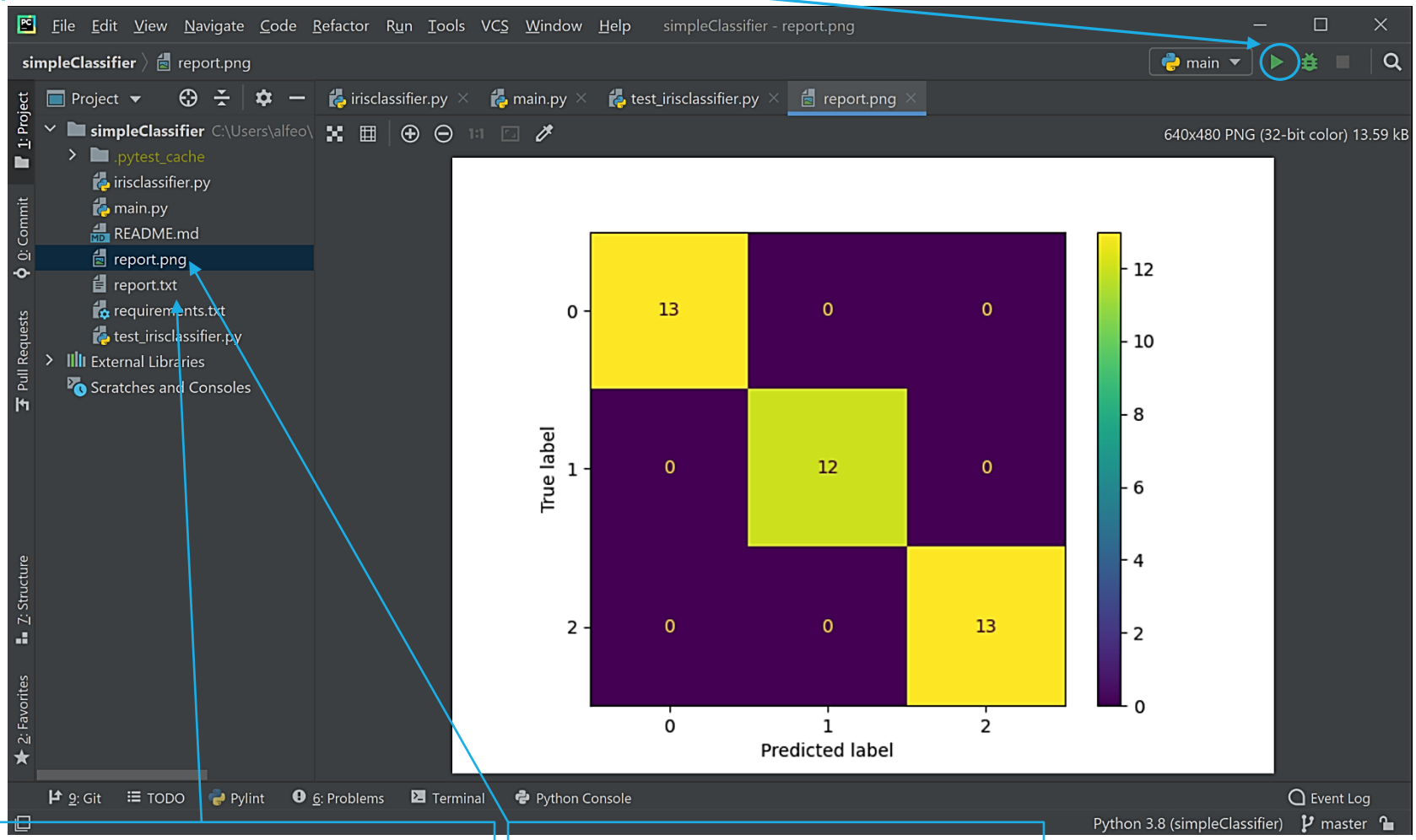
3. Select “main.py” in your project folder.



4. Click OK

RUN IT!

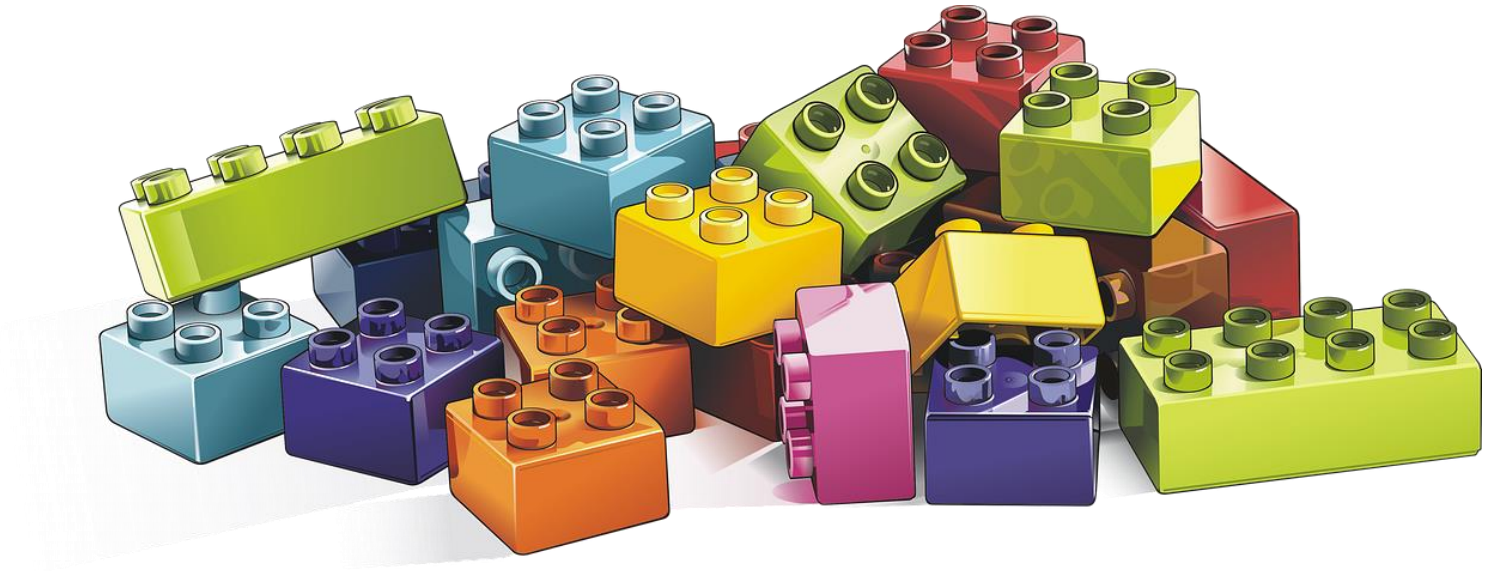
Click RUN



Txt with evaluation score!

The confusion matrix!

INTRODUCTION TO PACKAGES FOR YOUR PROJECT PIPELINE



Based on previous lecture by A. L. Alfeo

DESIGN A PROJECT PIPELINE IN PYTHON

So many packages available:

- Pylint: to write better code!
- Pandas: read data from a broad range of sources like CSV, SQL databases, and Excel files.
- Json and Joblib: to save/read models and configurations to/from files.
- Sci-kit learn: machine learning library with a broad range of clustering, regression and classification algorithms. It provides also data preprocessing, pipelining and performance analysis modules.
- Matplotlib: graphic visualization libraries.
- NumPy: to apply mathematical functions and scientific computation to multi-dimensional data

…and much more.

PROJECT ORGANIZATION

Consider that each class in your project should provide one or more functionalities that work with inputs and configurations to generate outputs. Inputs, configurations and outputs have to be stored in dedicated folders. Each class should be able to save/load data and configurations from those folders.

Let's start with a very simple example. We aim at classify a yearly set of time series. We need to create at least 3 folders:

- **data** -> tabular data, from raw (click to download) to segregated form
- **configs** -> configurations and models
- **results** -> outcomes, figures and scores obtained

UPLOAD/SAVE TABULAR DATA WITH PANDAS.DATAFRAME

```
from pandas import read_csv

# Read csv
data1 = read_csv('data/hotspotUrbanMobility-1.csv')
# Print dataframe shape
print(data1.shape)
# Read csv
data2 = read_csv('data/hotspotUrbanMobility-2.csv')
# Print dataframe shape
print(data2.shape)
# Append dataframes
data1 = data1.append(data2, ignore_index=True)
print(data1.shape)
# Save the new datafame as csv
data1.to_csv('data/completeDataset.csv', index=False)
```

Path of the csv to read

Dataframe shape consist of its numer of rows and column

Append 2 dataframes and ignore the indexes that pandas provides

Save the dataframe

DATAFRAME MANIPULATION WITH PANDAS.DATAFRAME

```
from pandas import read_csv

# Read csv
data = read_csv('data/completeDataset.csv')
# Check the content of the dataframe
print(data.describe())
# Remove column with constant values
data = data.drop('h24', axis=1)
# Remove anomalous instances
data = data.loc[data['Anomalous'] < 1]
print(data.describe())
# Save as csv
data.to_csv('data/preprocessedDataset.csv', index=False)
```

Provides statistics for each column in the dataframe

Drop the column (axis=1) whose label is 'h24'

Loc select instances by using a boolean array.

TRY THE LAST 2 SLIDES YOURSELF!

```
(198, 28)
(167, 28)
(365, 28)
        Anomalous     Cluster         Day  ...         h22         h23    h24
count  365.000000  365.000000  365.000000  ...  365.000000  365.000000  365.0
mean     0.101370    0.569863   15.720548  ...    0.472210    0.384617    1.0
std      0.302232    0.729045    8.808321  ...    0.373362    0.373992    0.0
min      0.000000    0.000000    1.000000  ...    0.000000    0.000000    1.0
25%      0.000000    0.000000    8.000000  ...    0.137218    0.130619    1.0
50%      0.000000    0.000000   16.000000  ...    0.403655    0.225237    1.0
75%      0.000000    1.000000   23.000000  ...    0.998552    0.783687    1.0
max      1.000000    2.000000   31.000000  ...    1.000000    1.000000    1.0

[8 rows x 28 columns]
       Anomalous     Cluster         Day  ...         h21         h22         h23
count      328.0  328.000000  328.000000  ...  328.000000  328.000000  328.000000
mean         0.0    0.530488   15.640244  ...    0.678014    0.455213    0.370639
std          0.0    0.716121    8.654346  ...    0.279946    0.371232    0.374739
min          0.0    0.000000    1.000000  ...    0.000000    0.000000    0.000000
25%          0.0    0.000000    8.000000  ...    0.467768    0.134583    0.129144
50%          0.0    0.000000   15.500000  ...    0.702473    0.381617    0.222655
75%          0.0    1.000000   23.000000  ...    1.000000    0.998032    0.774066
max          0.0    2.000000   31.000000  ...    1.000000    1.000000    1.000000

[8 rows x 27 columns]

Process finished with exit code 0
```

MORE ON TABULAR DATA

- Load csv as a dataframe, save a dataframe as csv
- Shape, head and quick description of a dataframe
- Dataframe concatenation and append
- Data sort by index or by value
- Data resampling and augmentation
- Dataframe selection by value or index
- Dataframe columns/row addition and elimination
- Replace by value, drop duplicate rows, group by value
- Drop NAN values , fill NAN value
- Data interpolation and signal smoothing via moving average

DATA SEGREGATION WITH PANDAS AND SCIKIT-LEARN

```
from pandas import read_csv
from sklearn.model_selection import train_test_split

# Read csv
data = read_csv('data/preprocessedDataset.csv')
print(data.describe())
# Random split 75% as training and 25% as testing
training_data, testing_data, training_labels, testing_labels =
                        train_test_split(data.iloc[:, 4:len(data.columns)], data.iloc[:, 1])
# Save as csv
training_data.to_csv('data/trainingData.csv', index=False)
testing_data.to_csv('data/testingData.csv', index=False)
training_labels.to_csv('data/trainingLabels.csv', index=False)
testing_labels.to_csv('data/testingLabels.csv', index=False)
```

Split randomly 75% and 25% of the dataset (default), once it knows which are the target and the data.

iloc provides integer-location based indexing

Scikit-learn allows you also to discretize and normalize your data via minmax or standard scaler procedure. **Those are very important if you are going to use an artificial neural network!**

MODEL DESIGN 1\2

Scikit-learn provides a number of different algorithms for **machine learning**.

A **classifier** is useful if you need to predict a given class (the label can also be a number).

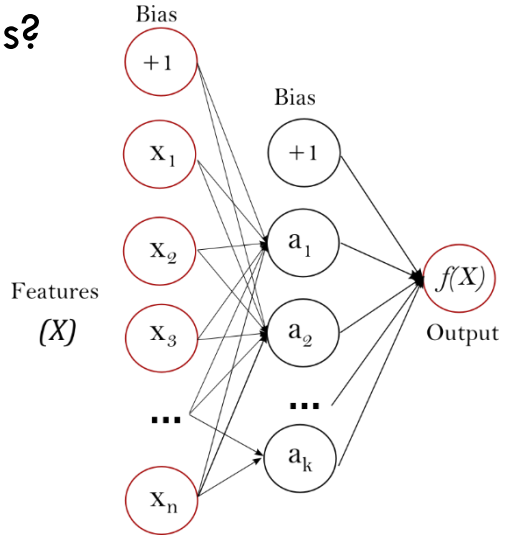
A well-known (not the only one) machine learning classifier are **artificial neural networks**. Scikit-learn provide a useful interface to use them.

```
from sklearn.neural_network import MLPClassifier
from pandas import read_csv
from numpy import ravel
# Read the data
training_data = read_csv('data/trainingData.csv')
training_labels = read_csv('data/trainingLabels.csv')
testing_data = read_csv('data/testingData.csv')
testing_labels = read_csv('data/testingLabels.csv')
# Build and train the classifier
mlp = MLPClassifier('max_iter'=100) .fit(training_data, training_labels)
mlp.predict(testing_data) # produce the predictions
mlp.score(testing_data, testing_labels) # compute the accuracy of the model
```

MODEL DESIGN 2\2

However, there is a number of **hyperparameters** to tune, such as:

- **Hidden layer number and size**: which values?
- **Activation function**: relu, tan, sigmoid, …?
- **Alpha**: what value?
- **Solver**: adam, sgd, …?
- **Batch size**: what value?
- **Max iterations**: what value?



Sometimes some recommendation can help you, but not always…

FROM JSON TO OBJECT

```
import json

class Params():
    def __init__(self, M):
        self.max_iter = M

best_par = Params(60)
out_file = open("config/modelConfiguration.json", "w")
json.dump(best_par, out_file)
out_file.close()

# load best configuration
config_path = "config/modelConfiguration.json"
with open(config_path, "r") as f:
    params = json.load(f)

params_object = Params(**params)
```

Use a simple class to keep the model's parameters. You can use json.dump() to store it in a file.

Load the JSON . Remember, it should be compliant to the expected json schema i.e. pass the validation

Create a new object, and pass the JSON dictionary as a map to convert JSON data into a custom Python Object

HYPER-PARAMETRIZATION VIA GRID SEARCH

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from pandas import read_csv
from numpy import ravel
import json

# Read the data
training_data = read_csv('data/trainingData.csv')
training_labels = read_csv('data/trainingLabels.csv')
# setup Grid Search for MLP
mlp = MLPClassifier()
# values to test
parameters = {'max_iter': (100, 200, 300)}
# apply grid search
gs = GridSearchCV(mlp, parameters)
gs.fit(training_data, ravel(training_labels))
# save best configuration
config_path = 'config/modelConfiguration.json'
with open(config_path, 'w') as f:
    json.dump(gs.best_params_, f)
```

If "cv=None", GridSearch uses Cross-fold validation. Explicitly pass test and training sets with PredefinedSplit

The ML model

Max_iter is the name of the hyperparameter of MLPclassifier

Use ravel to transform dataframe to narray

Save the best hyperparameter as JSON

ARCHITECTURE EVALUATION

```
from numpy import ravel
from pandas import read_csv
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
import json

# Read the data
training_data = read_csv('data/trainingData.csv')
training_labels = read_csv('data/trainingLabels.csv')
testing_data = read_csv('data/testingData.csv')
testing_labels = read_csv('data/testingLabels.csv')
# load best configuration
config_path = "config/modelConfiguration.json"
with open(config_path, "r") as f:
    params = json.load(f)
# train the model
model = MLPClassifier(**params).fit(training_data, ravel(training_labels))
# compute labels on testing data
labels = model.predict(testing_data)
# evaluate accuracy score
score = accuracy_score(ravel(testing_labels), labels)
```

You can use different performances measures

MODEL DEPLOYMENT WITH JOBLIB

```
from pandas import read_csv
from sklearn.neural_network import MLPClassifier
from numpy import ravel
import joblib

# Read the training data
training_data = read_csv('data/trainingData.csv')
training_labels = read_csv('data/trainingLabels.csv')
# train the model
initial_model = MLPClassifier(random_state=0).fit(training_data, ravel(training_labels))
# save the model
joblib.dump(initial_model, 'config/fitted_model.sav')
#[...]
# Read new data
testing_data = read_csv('data/testingData.csv')
testing_labels = read_csv('data/testingLabels.csv')
# load the model
model = joblib.load('config/fitted_model.sav')
# Evaluate the initial model on new data
print('Evaluation score:')
print(model.score(testing_data, ravel(testing_labels)))
```

Random_state for reproducibility

Save a fitted model to file

Load a fitted model from file

Use the model with new data

TRY IT YOURSELF!

```
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from pandas import read_csv
from numpy import ravel
import json

data1 = read_csv('data/hotspotUrbanMobility-1.csv')
data2 = read_csv('data/hotspotUrbanMobility-2.csv')
data = data1.append(data2, ignore_index=True)
data = data.drop('h24', axis=1)
data = data.loc[data['Anomalous'] < 1]
data.to_csv('data/preprocessedDataset.csv', index=False)
data = read_csv ('data/preprocessedDataset.csv' )
training_data, testing_data, training_labels, testing_labels =
                    train_test_split(data.iloc[:,
                    4:len(data.columns)], data.iloc[:, 1])
mlp = MLPClassifier(random_state=0)
parameters = {'max_iter': (100, 200, 300)}
gs = GridSearchCV(mlp, parameters)
gs.fit(training_data, ravel(training_labels))
config_path = 'config/modelConfiguration.json'
with open(config_path, 'w') as f:
   json.dump(gs.best_params_, f)
with open(config_path, "r") as f:
   params = json.load(f)
model = MLPClassifier(**params).fit(training_data,
                    ravel(training_labels))
labels = model.predict(testing_data)
score = accuracy_score(ravel(testing_labels), labels)
print(score)
```

What's the accuracy score of your best model?

MORE ON MODEL EVALUATION…

- Choose the right performance metric
- K-Folds and Stratified-K-Fold cross-validation
- How much time does an instruction take to execute?
- Performance visualization with scikit-learn, and how to write them on files

…AND GRAPHS WITH MATPLOTLIB.PYPLOT

Create beautiful graphs (here you have the cheatsheets):

- Plot and save a figure
- Histogram and barplot
- Boxplots
- Scatterplot
- Create single and multiple graphs

REST API AS INTERFACES 1\3

REST APIs are becoming the standard de-facto for machine-to-machine communication, and can be used to communicate among the different modules of your project. For instance you can:

- Generate some end-points, one for each functionality to expose
- Use POST method to create a new resource (e.g. build the training set, pre-process it)
- Use GET method to retrieve a resource from different components of the application (e.g. the configuration file, or even the JSON version of a configuration object)

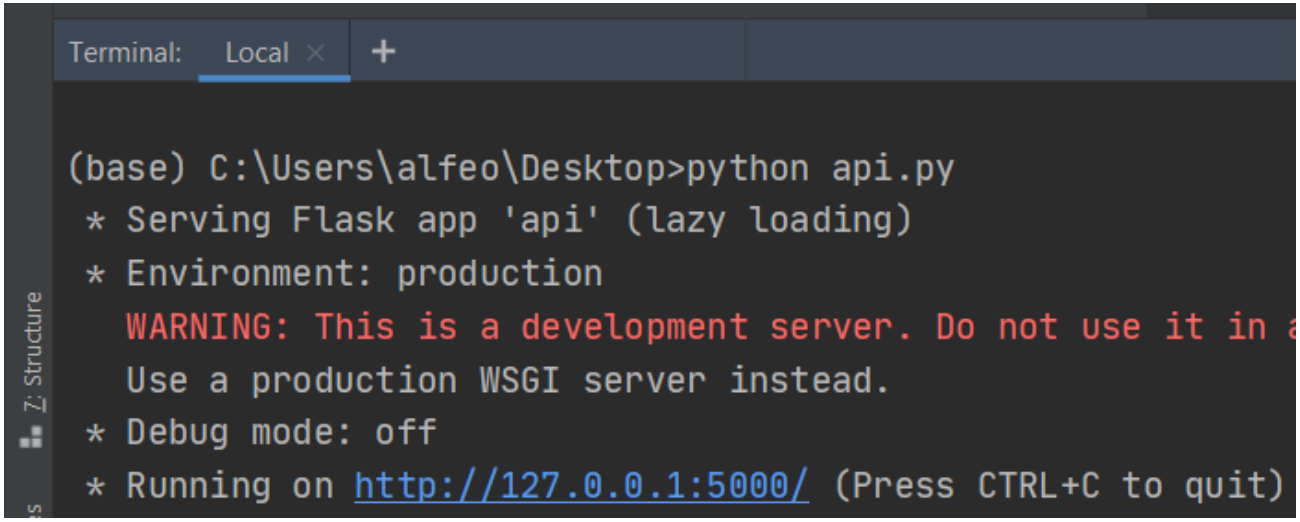
REST API AS INTERFACES 2\3

In Python you can use Flask and Request to build and use a simple REST server. For instance we can define an end-point *test* to check if the server is alive:

```
from flask import Flask, Response

app = Flask(__name__) # always use this

@app.route("/test", methods=['GET'])
def test():
    # This is a test endpoint used to check if the server is running
    return Response("<h1>I'm alive!</h1>",status=200, mimetype='text/html')

app.run()
```

REST API AS INTERFACES 3\3

Save it as *api.py*. From Pycharm terminal, move into *api.py* 's location and run it

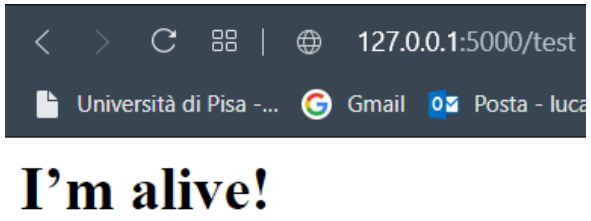


TRY IT YOURSELF!

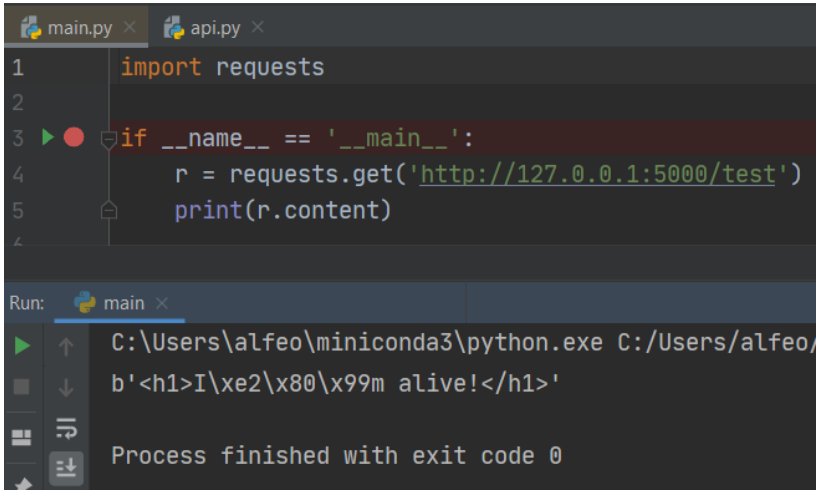
Check on

http://127.0.0.1:5000/test

via browser…



…or via request



IMPROVE YOUR TEAMWORK!

Based on previous lecture by A. L. Alfeo

CODE QUALITY: PEP8 STANDARD

The primary focus of **PEP8** (Python Enhancement Proposal) standard is to improve the readability and consistency of Python code. Examples:

Use 4 spaces per indentation level. Use spaces, not tabs.

Limit all lines to a maximum of 79 characters.

Imports should be as specific as possible

Surround top-level functions and classes with two blank lines.

Surround method definitions inside classes with a single blank line.

Use blank lines sparingly inside functions to show clear steps.

Surround the following binary operators with a single space on either side:

- Assignment operators (=, +=, -=, and so forth)
- Comparisons (==, !=, >, <. >=, <=) and (is, is not, in, not in)
- Booleans (and, not, or)

PEP8 - NAMING

Function: Use a lowercase word or words. Separate words by underscores to improve readability. I.e. function, my_function

Variable: Use a lowercase single letter, word, or words. Separate words with underscores to improve readability. I.e. x, var, my_variable

Class: Start each word with a capital letter. Do not separate words with underscores. This style is called camel case. I.e. Model, MyClass

Method: Use a lowercase word or words. Separate words with underscores to improve readability. I.e. class_method, method

Constant: Use an uppercase single letter, word, or words. Separate words with underscores to improve readability. I.e. CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT

Module: Use a short, lowercase word or words. Separate words with underscores to improve readability. I.e. module.py, my_module.py

Package: Use a short, lowercase word or words. Do not separate words with underscores

AUTOMATIC QUALITY CHECKING

Pylint is a source-code, bug and quality checker for the Python programming language. It checks:

if the code is compliant with Python's **PEP8** standard

if each module is properly imported and used

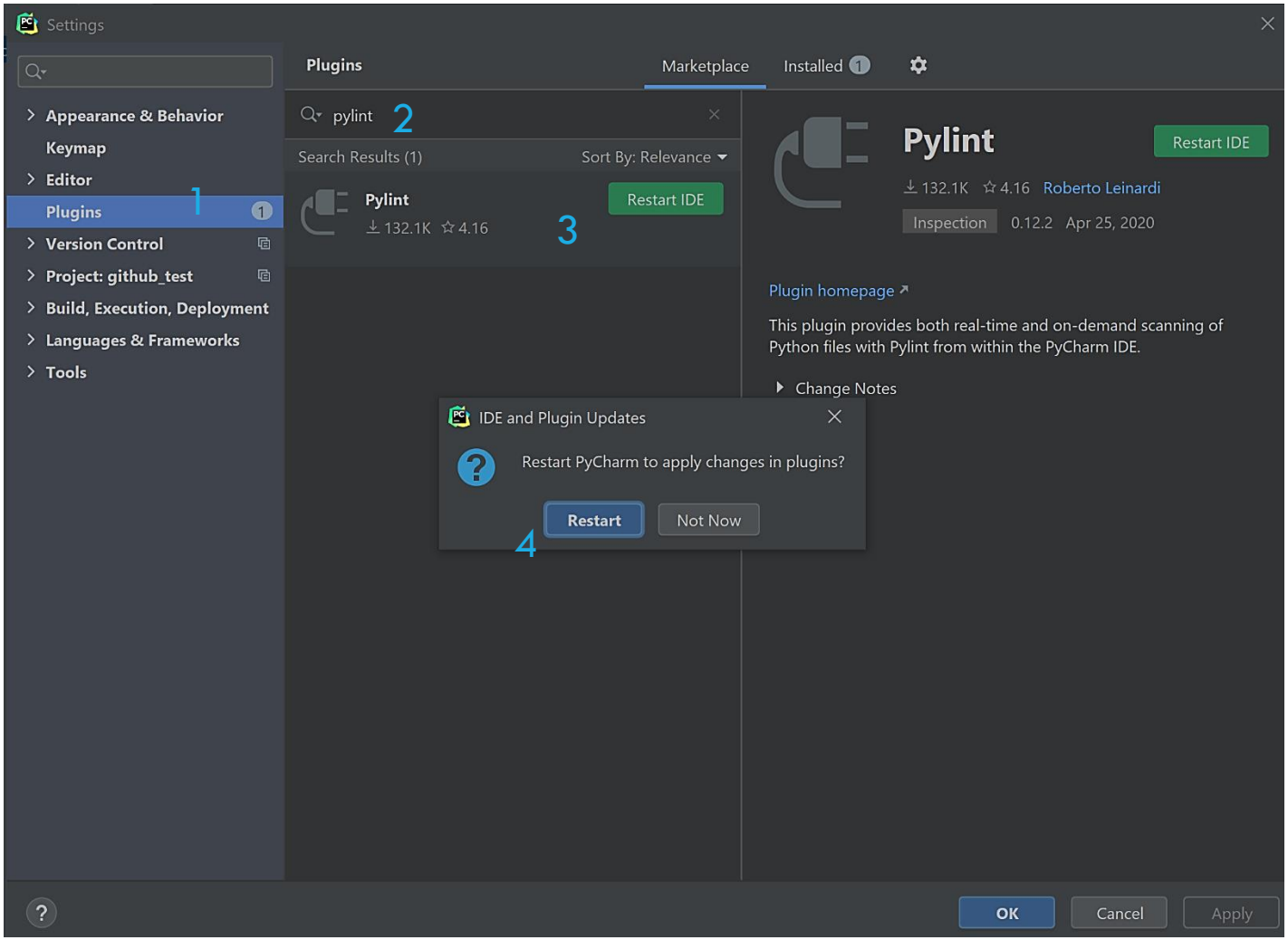
if declared interfaces are truly implemented

if there is duplicated code

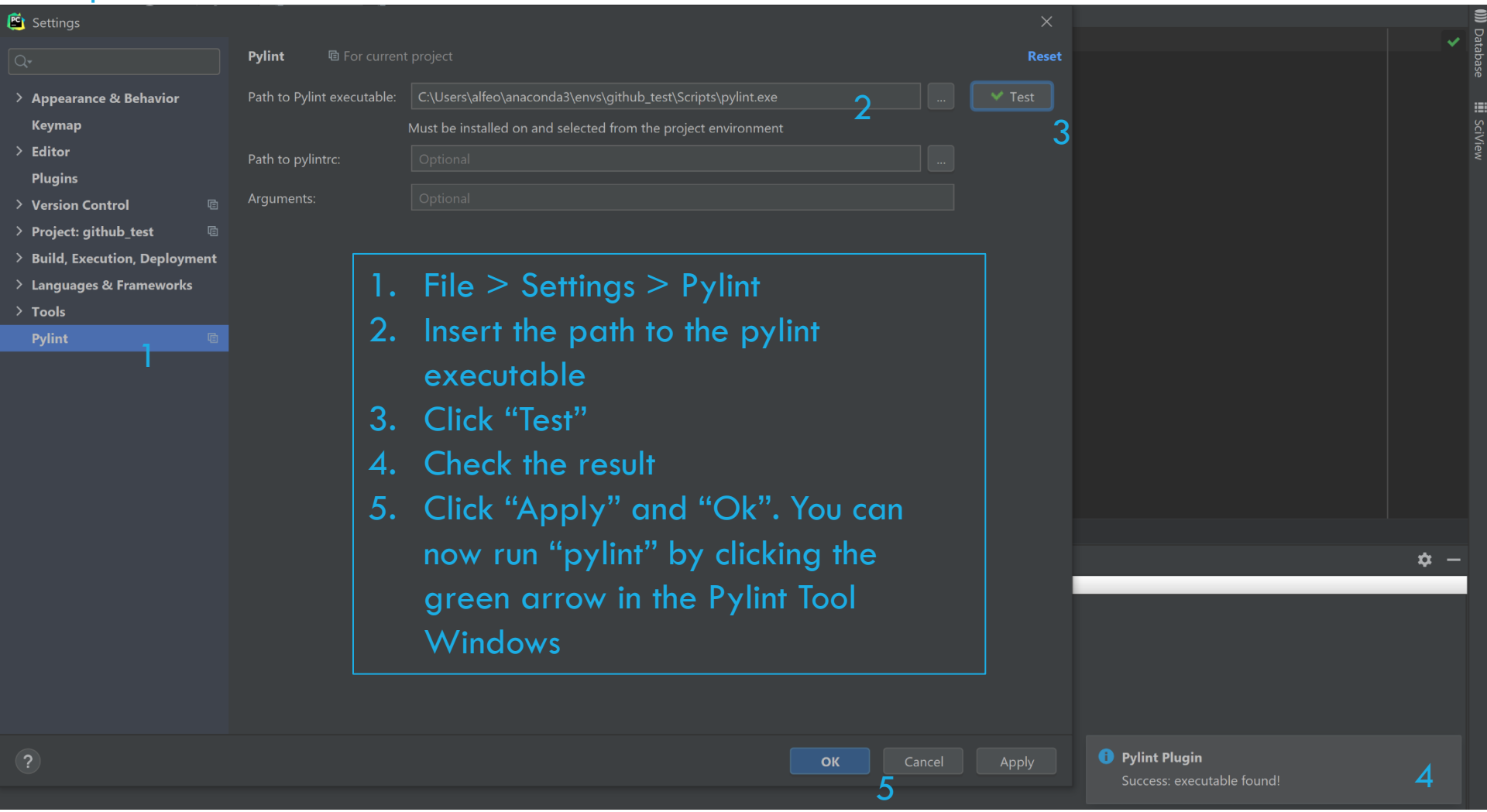
Pylint plugin is fully customizable: modify your *pylintrc* to customize which errors or conventions are important to you.

PYLINT PLUGIN INSTALLATION

1. File > Settings > Plugin
2. Search for “pylint”
3. Install Pylint plugin
4. Restart Pycharm

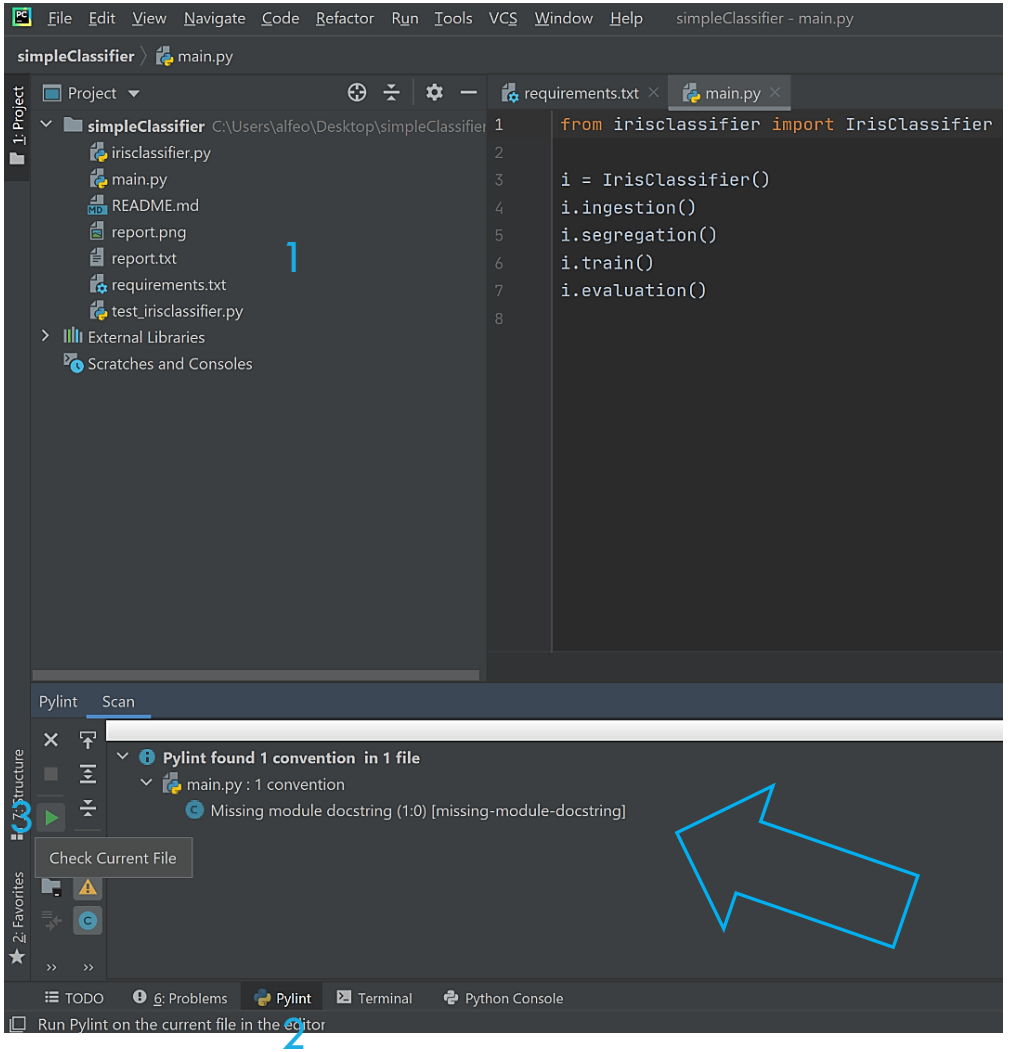


PYLINT PLUGIN SETUP

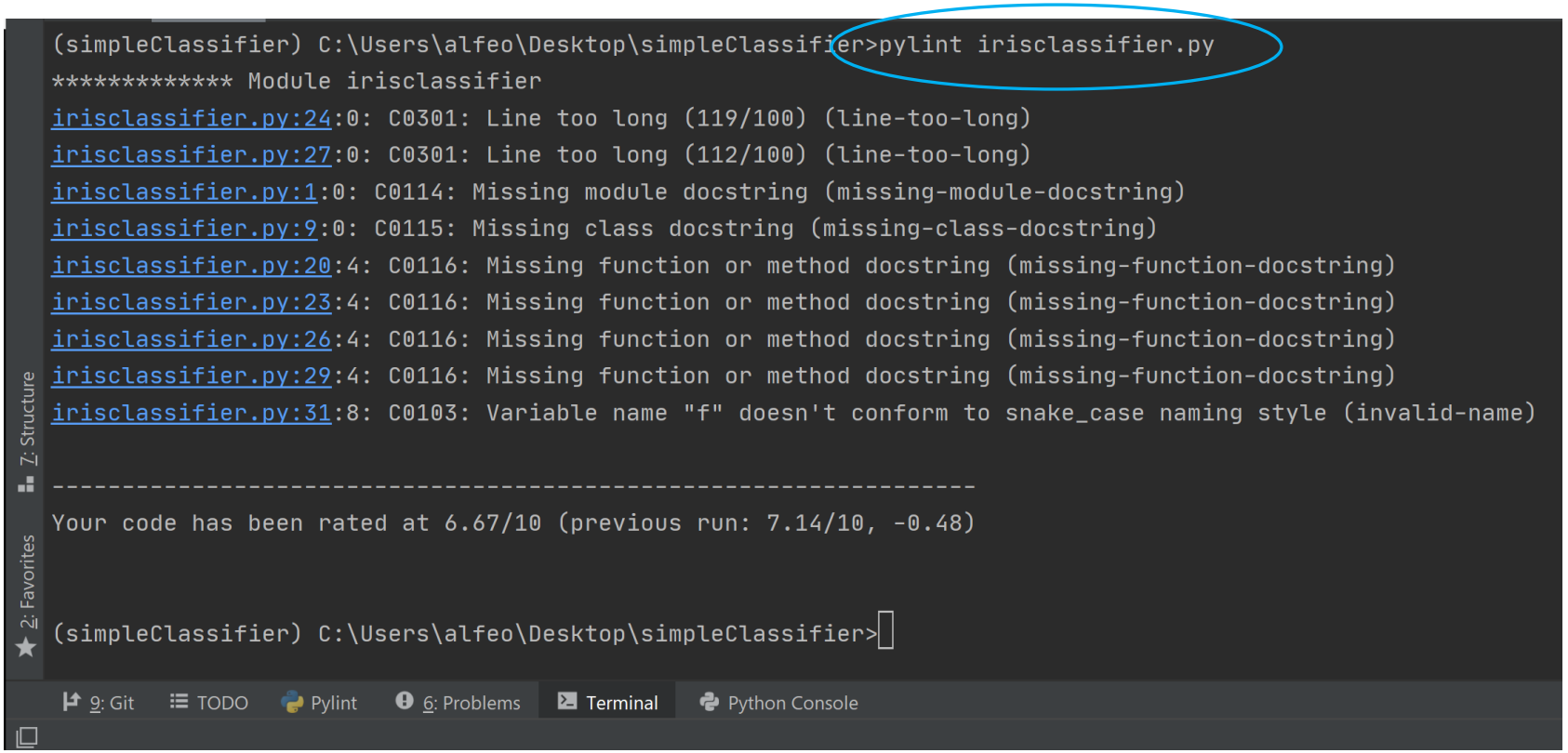


PYLINT TOOL WINDOW USAGE

1. Double click on a Python file in the Project View
2. Open the Pylint Tool Windows
3. Click the green arrow



TRY PYLINT VIA COMMAND LINE

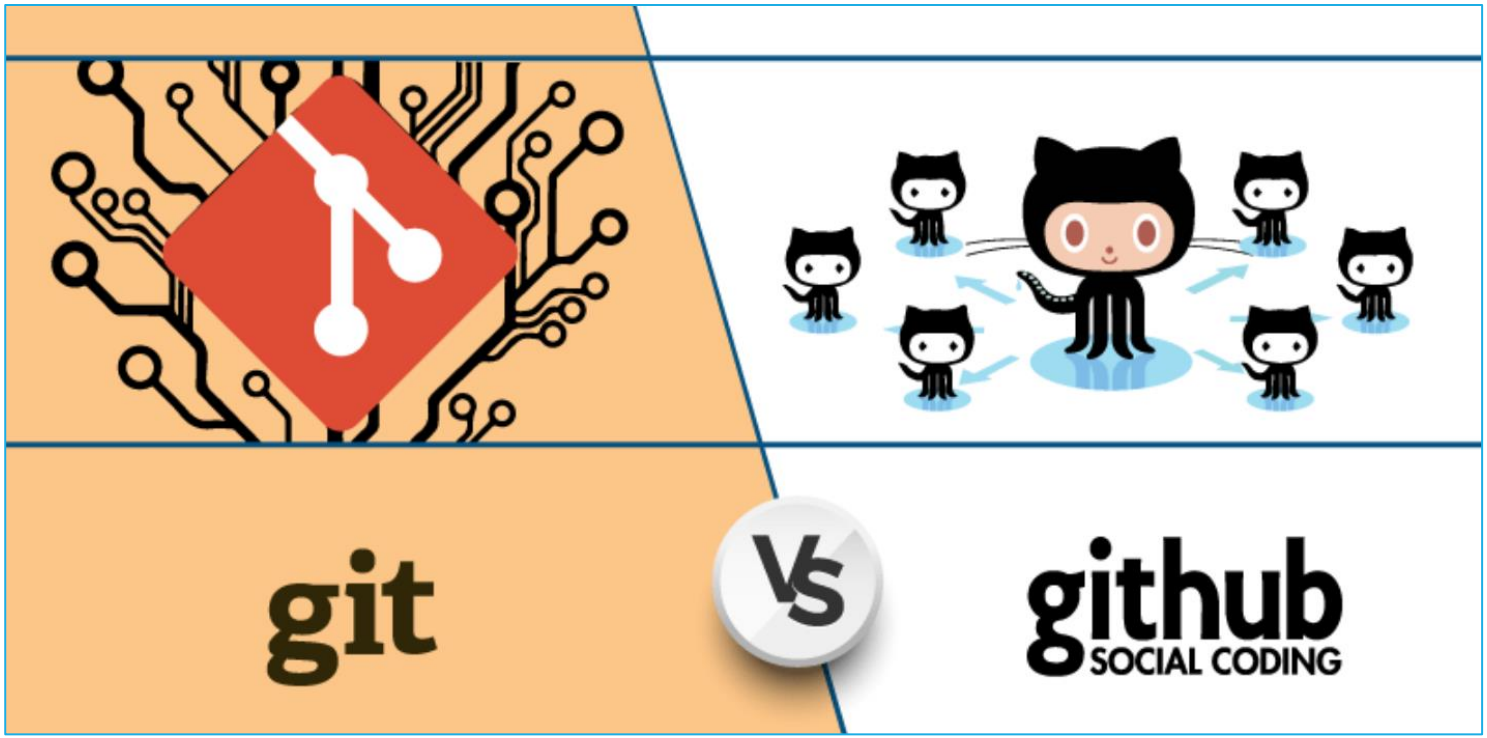


MORE ON CODING ASSISTANCE

Pycharm provides a number of functionalities aimed at helping you while writing your code, such as:

- **Python refactoring**:
  - renaming
  - extract methods
  - introduce variables and constants
- **Coding assistance and analysis**:
  - code completion
  - syntax and error highlighting
  - quick fixes
  - guided import
  - integrated Python debugger
  - code testing and code coverage

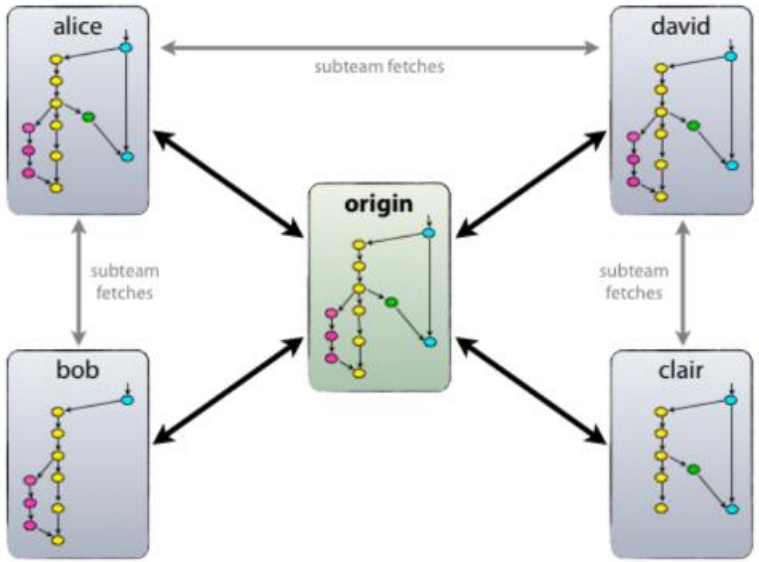
GIT AND GITHUB



Based on the lecture of F. Galatolo

GIT

Git is a **distributed Version Control System** in which each partecipant can have different view of the project.



REPOSITORY

A Repository (or "**repo**") is a data structure containing all project's **files and history**. You can **clone** a remote repo with the clone command:

```
git clone <repo url>
```

In a repo some files starting with ".git", those are special files and folders used to store the project history and to instruct git.

.git folder containing the project history (do not touch!)

.gitignore file containing ignoring rules

.gitmodules file containing submodules information

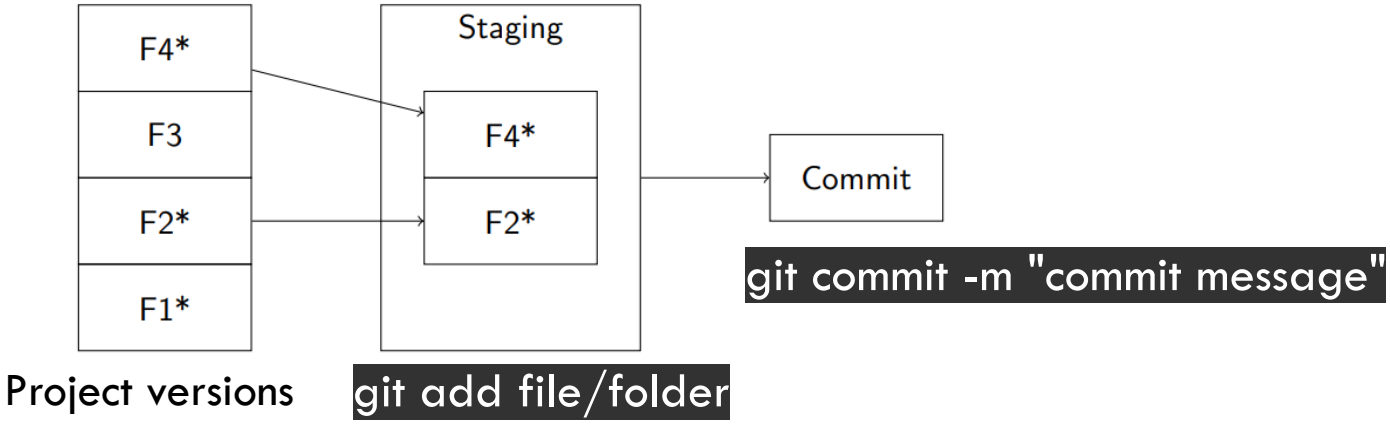
COMMIT 1/2

A commit is a snapshot of the project in a given time, it is uniquely identified by its hash and can be commented by a message.

Commits are immutable and represent a transition from a state to another, i.e. atomic units of modification within a project.

A git commit is a two-stages process.

- Add files to the staging environment
- Commit the changes



COMMIT 2/2

HEAD is a Git variable that points to the most recent commit.

HEAD can be changed as it follows

```
git reset <NEW_HEAD>
```

If you want to change the HEAD and the files in your repo, use

```
git reset --hard <NEW_HEAD>
```

Tags allows pointing to specific commits that represent milestones.

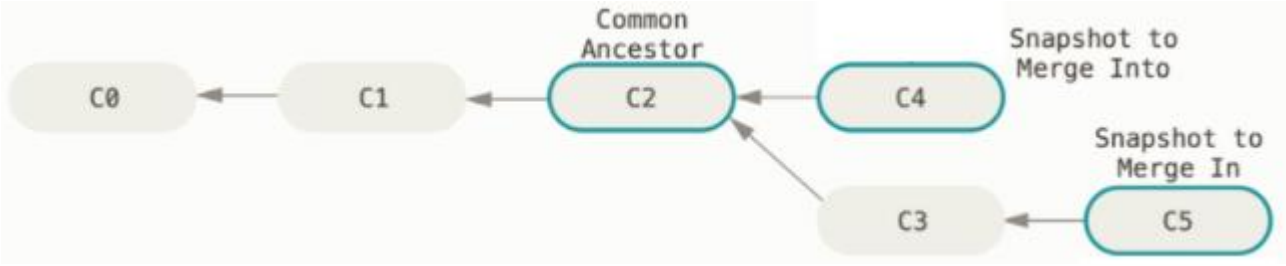
BRANCH

A **branch** is a semantically significant collection of commits. A commit is **always** in a branch. As a commit represent an atomic unit of modification, a branch represent a continuous flow of modifications.

- `git branch`
  - Show all the existing branches and the active one (denoted with an asterisk)
- `git branch <branch>`
  - Create a new branch called <branch>
- `git checkout <branch>`
  - Switch working on the branch <branch>
- `git checkout -b <branch>`
  - Create <branch> and switch working on it

MERGE

Different branches (or even single commits) can be **merged** in order to transfer all the changes developed in different branches.



All you have to do is **check out** the branch you wish to merge into and then run the git merge command:

- `git merge <branch>`
  - Merges <branch> in the active branch
- `git merge –abort`
  - Abort the merge and restore everything as it was before git merge

HANDLING MERGE CONFLICTS

If the commits changed different files or different sections of the same files the merge is **without conflicts**.

If the commits changed the same sections of the same files (at least once) the commit is **with conflict**.

Handling conflicts is pretty easy. When a conflict is detected git puts in the place of the conflict:

```
Some non-conflicting text
<<<<<<< HEAD
CONFLICTING PORTION COMING FROM HEAD
=======
CONFLICTING PORTION COMING FROM bugfix
>>>>>>> bugfix
Some other non-conflicting text
```

Is up to you keep the portions that you want and then git add and git commit the changes

REMOTES

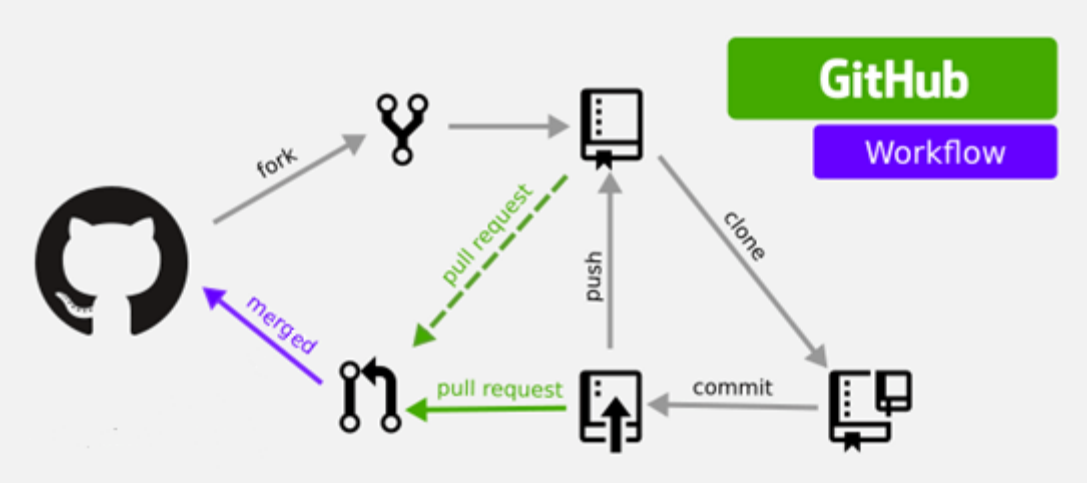
Git is distributed. A **remote** is a reference to a remote instance of the repository. The default remote is called **origin**: if you clone a repository then origin points to the cloned repo.

- `git fetch <remote> <branch>`
  - Fetch the commits from branch <branch> of <remote> in the local branch <remote>/<branch>
- `git pull <remote> <branch>`
  - Fetch the commits from branch <branch> of <remote> and merge them in the local <branch>
- `git push <remote> <branch>`
  - Push the state of the local branch to the origin branch <branch>

GITHUB

- GitHub is Git server.
- Used by over 65 Million Users
- Contains over 200 Million Repositories
- Largest source code host in the world

MORE ON GITHUB: FORK AND PULL



In GitHub any public repository can be **Forked**. After a fork you own an exact copy of that repository into yours (you are the administrator). A fork is not a Git clone.

If you want to ask for the integration of your changes in the forked repository you can create a **Pull Request**. In the pull request you have to specify the destination branch (original repository) and the source branch (your repository).

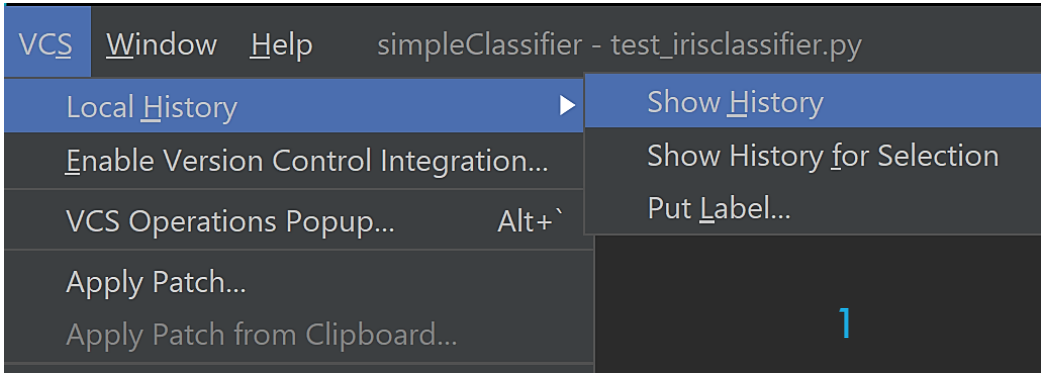
PYCHARM VCS

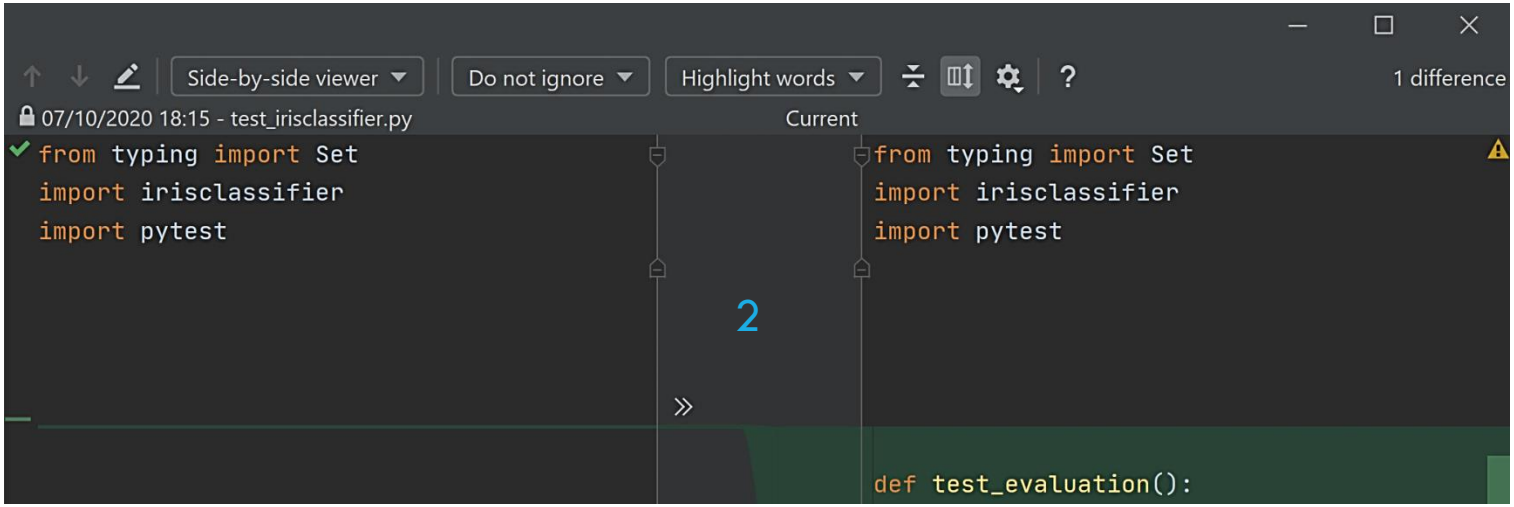
Based on previous lecture by A. L. Alfeo

BASIC VERSION CONTROL

In Pycharm the Version Control can be accessed locally:

1. via VCS > Local History
2. Stores only local and recent change in the project highlighting them

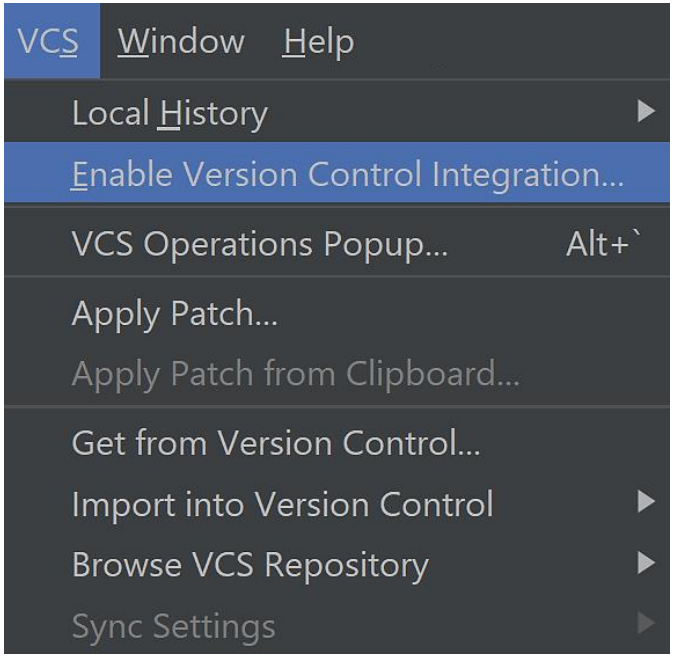


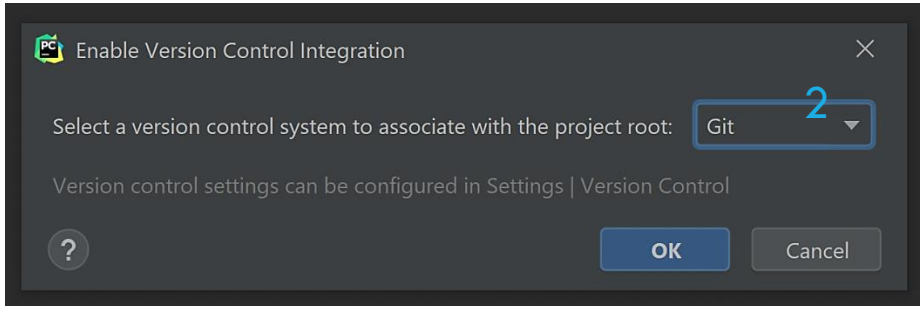


MANAGE PROJECTS WITH GITHUB

Github is way more convenient especially when a project is developed in group. Now you need to:

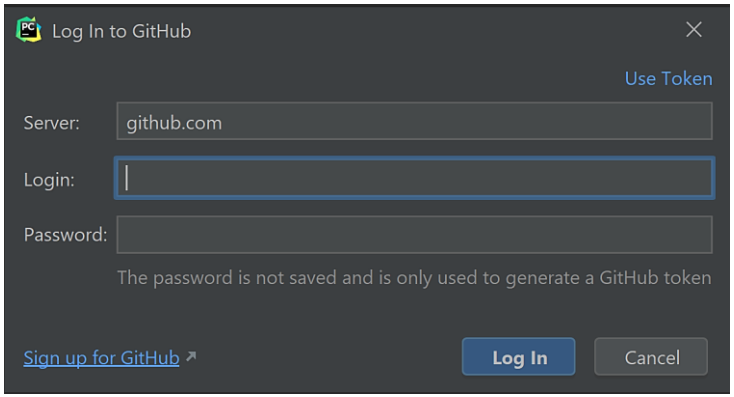
1. Install **Git** using only the default settings in the installation process
2. After VC integration is enabled (1 and 2), PyCharm will ask you whether you want to share project settings files via VCS.
3. In the dialog you can choose "**Always Add**" to synchronize project settings with other repository users who work with PyCharm.



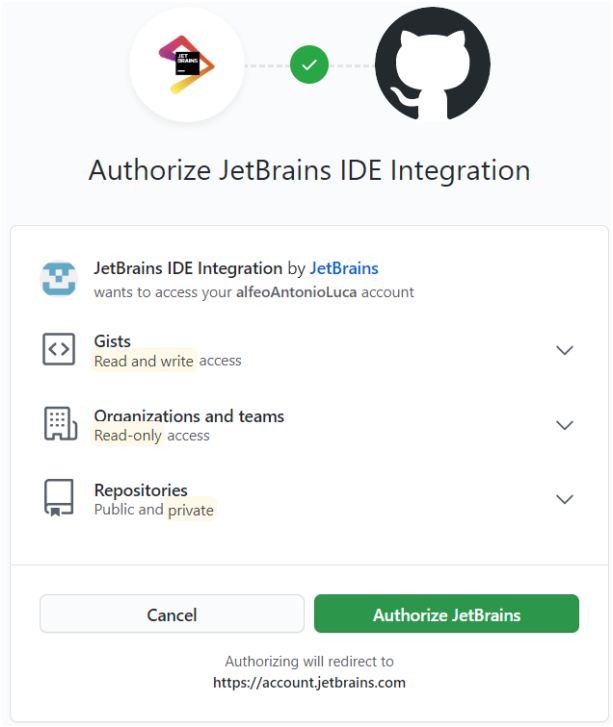


ENABLE VCI TO USE GITHUB

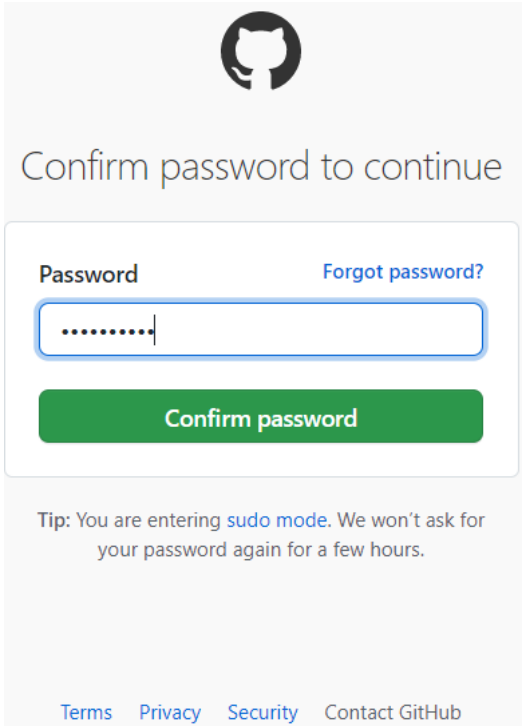
Login into GitHub, requesting a new access token with your login and password (1). Then, authorize JetBrain IDE integration (2 and 3).



1



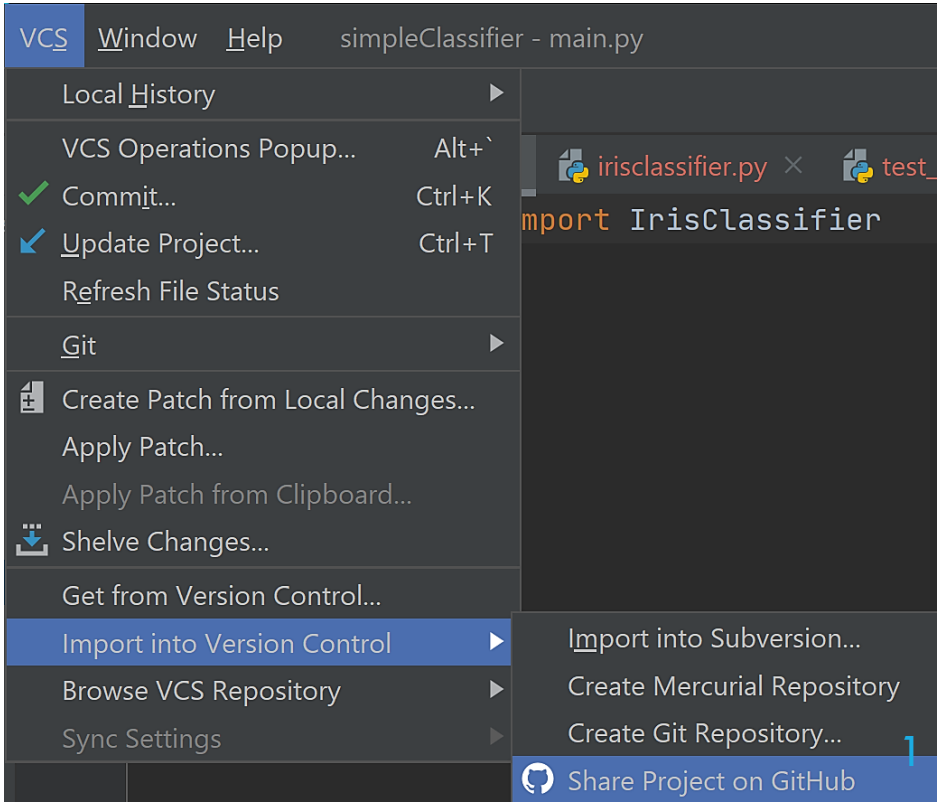
2

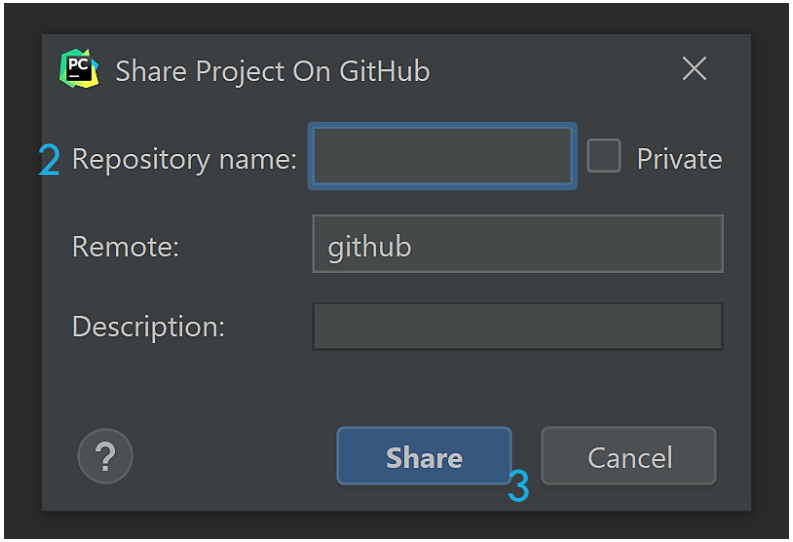


3

SHARE A PROJECT 1/2

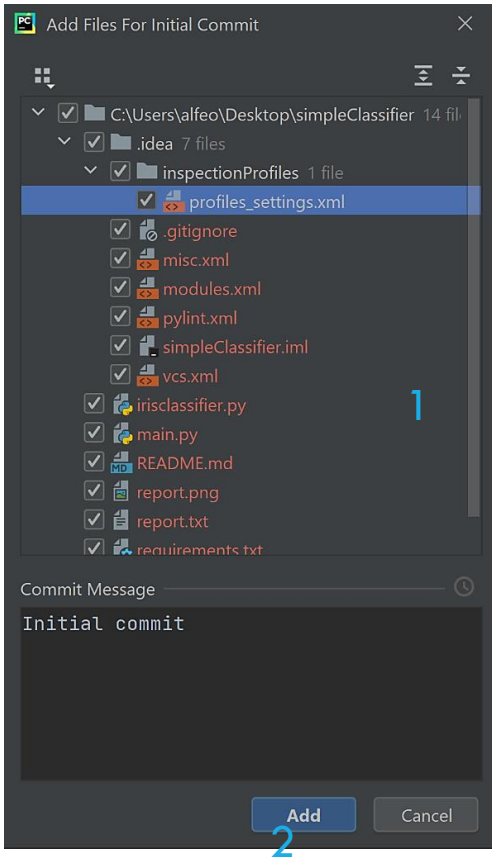
1. When connection to GitHub has been established, the Share Project on GitHub dialog appears (VCS > Import Version Control > Share …).
2. Name the repository and click share

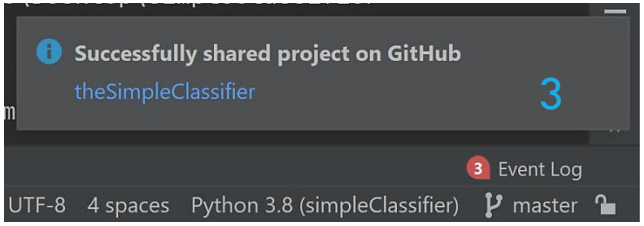




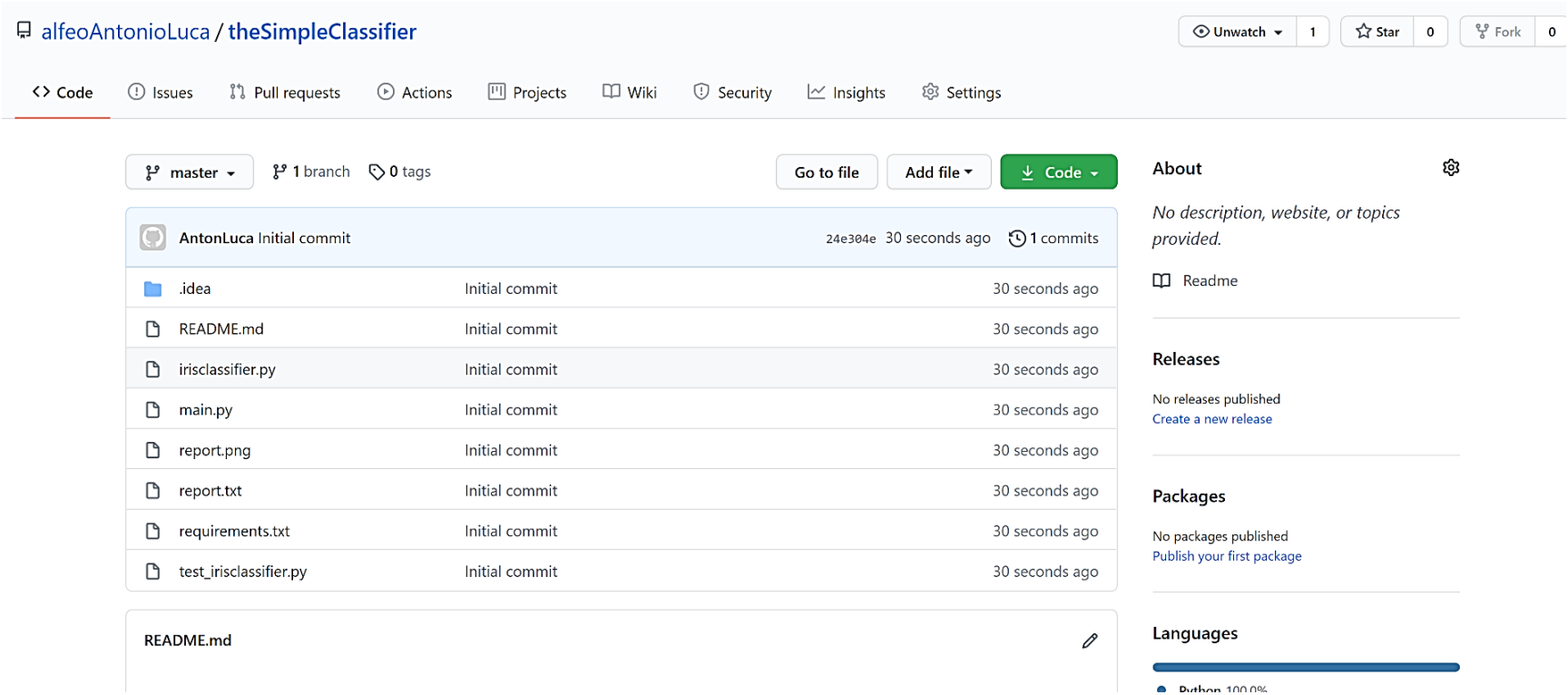
SHARE A PROJECT 2/2

1. Being the initial commit, of the project every file is highlighted in red (they are not in the VCS).
2. Click Add
3. PyCharm will notify once the process is finalized





… ON GITHUB



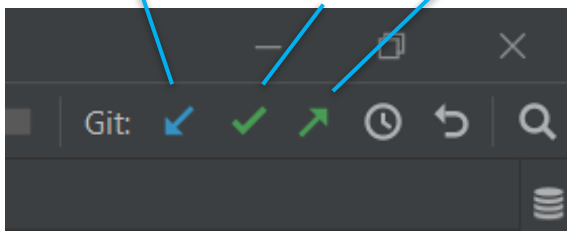
Let's say you have a **central repository** for your project, the contribution of each coder will be highlighted by the commits she/he made. In the next slide you will see how to "sign" your commit °

COMMIT AND PUSH

Update (pull)

Push

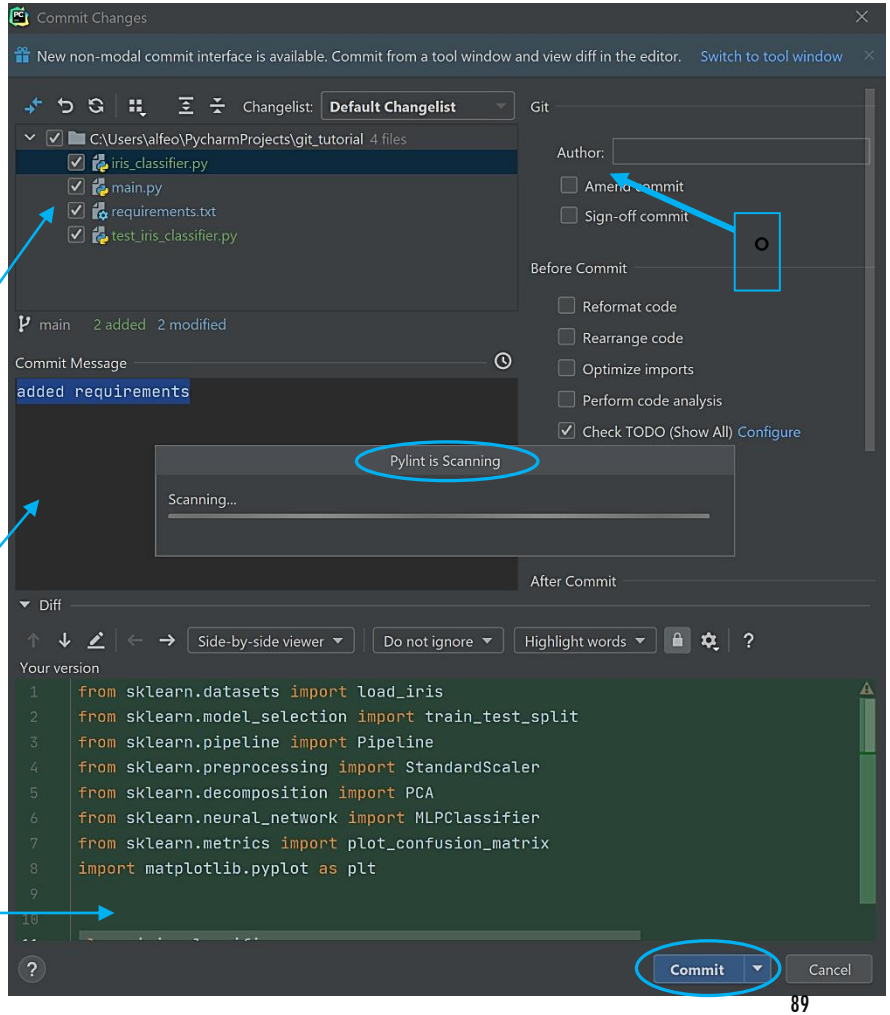
Commit



Color code:
- Blue: modified
- Green: commit done
- Red: not in the VCS
- White: push done

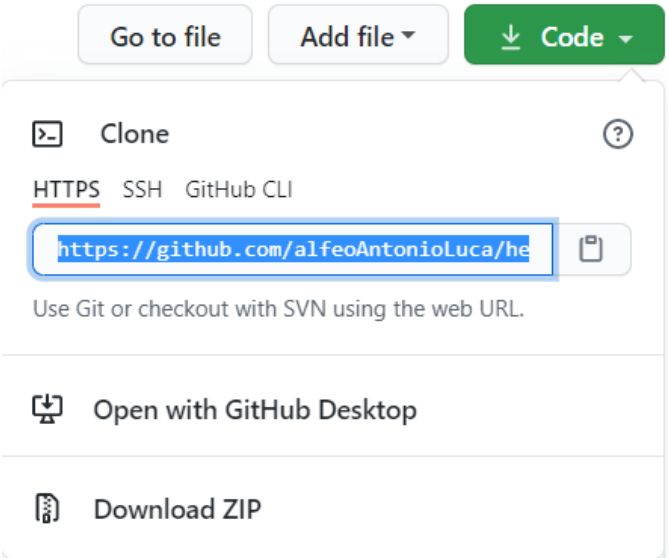
Explain clearly the change provided

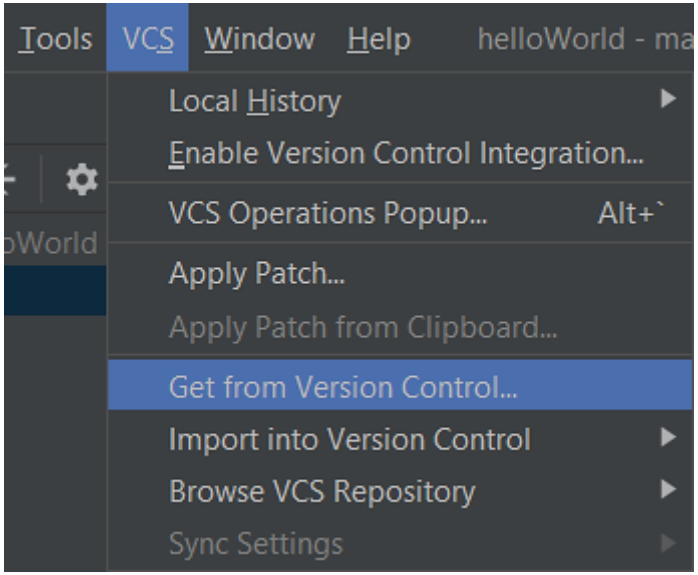
After the first commit here there will be the “before and after” comparison



CLONE FROM GIT 1/2

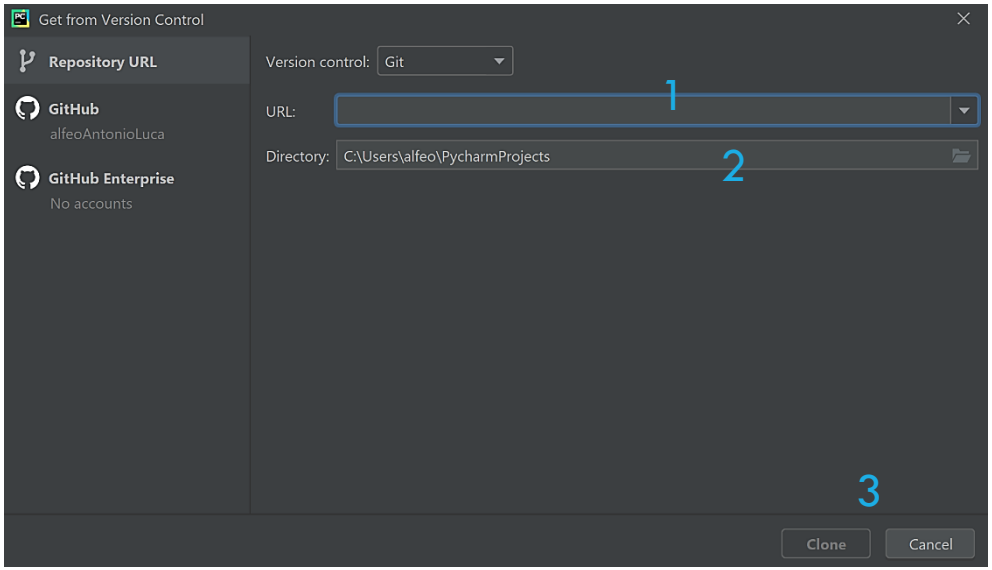
1. Go to GitHub and annotate the URL of the repository you are interested in.
2. From the main menu, choose VCS > Get from Version Control, and choose GitHub on the left.





CLONE FROM GIT 2/2

1. Specify the URL of the repository that you want to clone (the one you annotated from GitHub).
2. In the Directory field, enter the path to the folder where your local Git repository will be created.
3. Click **Clone**. If you want to create a project based on these sources, click Yes in the confirmation dialog. PyCharm will automatically set Git root mapping to the project root directory.



MORE ON PYCHARM VCS

PyCharm provides **version control integration**, a unified user interface for many VCS such as **GithHub**.

It supports your contribution to the project via commit, push, pull, change lists, cloning… and even branch, merge, and conflict via Git.

Here you have a couple of video-tutorials [1, 2, 3]