

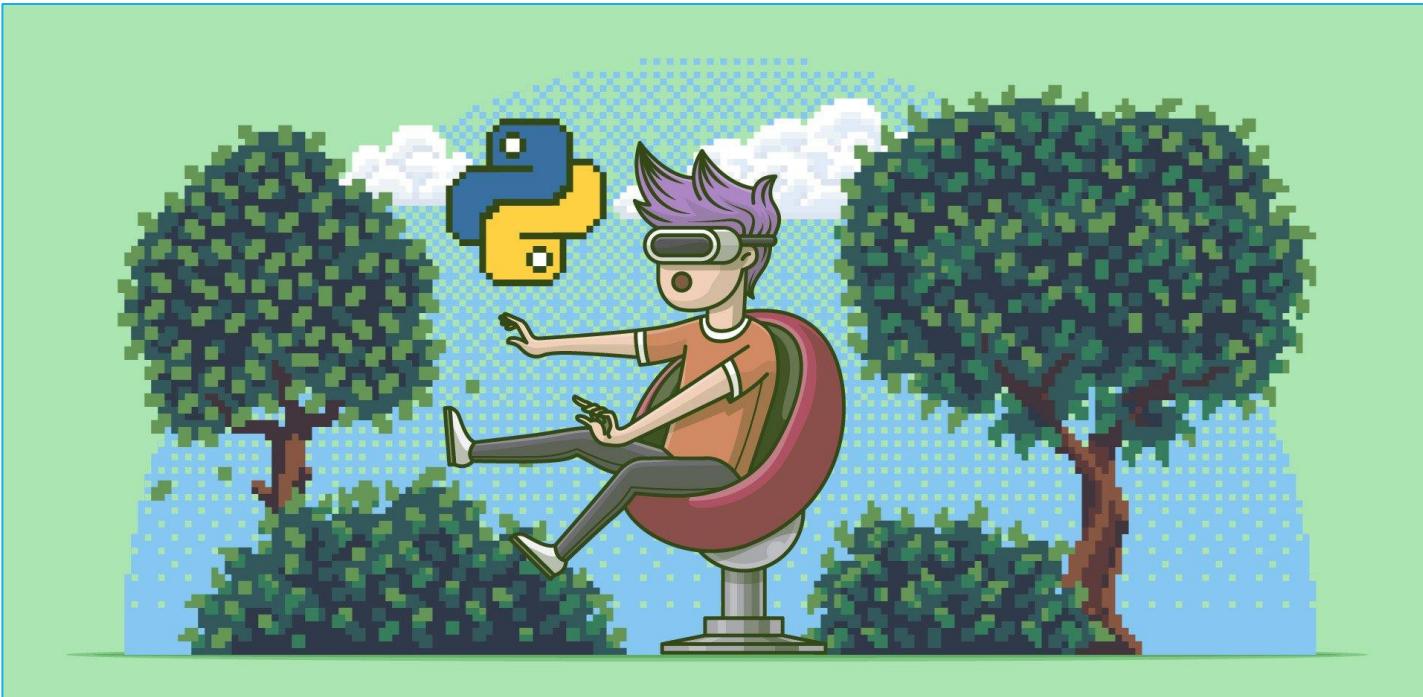
# USING PYCHARM, PYLINT, AND GITHUB TO DESIGN A PROJECT PIPELINE IN PYTHON

Antonio Luca Alfeo

# OVERVIEW

- |                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Introduction to Python and Virtual environment <ul style="list-style-type: none"><li>I. Package managing</li><li>II. Anaconda</li></ul>                                                                       | 5. Code quality checking <ul style="list-style-type: none"><li>I. PEP8</li><li>II. Pylint</li><li>III. Plugin installation and usage</li></ul>                                                                                                                                                                                                                                  | 7. REST API as interfaces <ul style="list-style-type: none"><li>I. Flask and Flask-restful</li><li>II. Sending request with arguments</li></ul>                                          |
| 2. Pycharm Installation                                                                                                                                                                                          | 6. Basics to design a structured project <ul style="list-style-type: none"><li>I. Workspace organization</li><li>II. Tabular data with pandas and sqlite</li><li>III. Data segregation with Sklearn</li><li>IV. Model design</li><li>V. From JSON to Object</li><li>VI. Model hyperparametrization</li><li>VII. Performance evaluation</li><li>VIII. Model deployment</li></ul> | 8. Introduction to Git and Github <ul style="list-style-type: none"><li>I. Repo</li><li>II. Commit</li><li>III. Branch</li><li>IV. Merge</li><li>V. Remotes</li><li>VI. Github</li></ul> |
| 3. Python basics <ul style="list-style-type: none"><li>I. Indentation</li><li>II. Types and Variables</li><li>III. Conditional Instructions</li><li>IV. Loops</li><li>V. Functions</li><li>VI. Classes</li></ul> | 9. Version Control on Pycharm <ul style="list-style-type: none"><li>I. Local history</li><li>II. Connect to GitHub</li><li>III. Github: share a project</li><li>IV. Github: commit and push</li><li>V. Github: clone a project</li></ul>                                                                                                                                        |                                                                                                                                                                                          |
| 4. Work with Pycharm projects <ul style="list-style-type: none"><li>I. Interpreter config</li><li>II. Requirements management</li><li>III. Project navigation and run</li></ul>                                  |                                                                                                                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                          |

# INTRODUCTION TO PYTHON

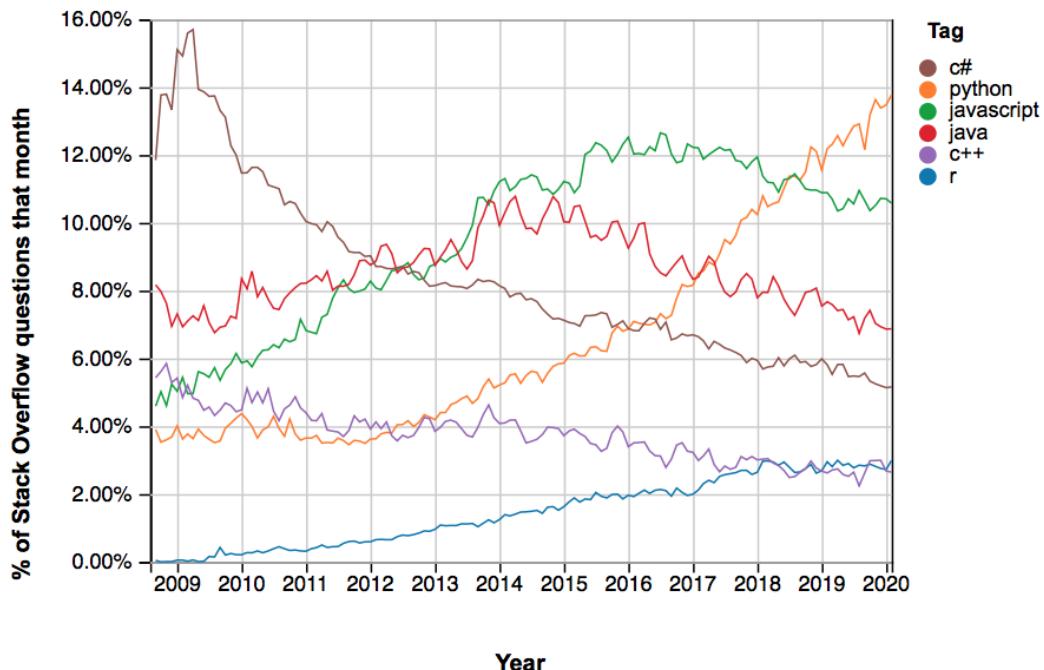


Based on previous lecture by [F. Galatolo](#)

# 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 require 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
```

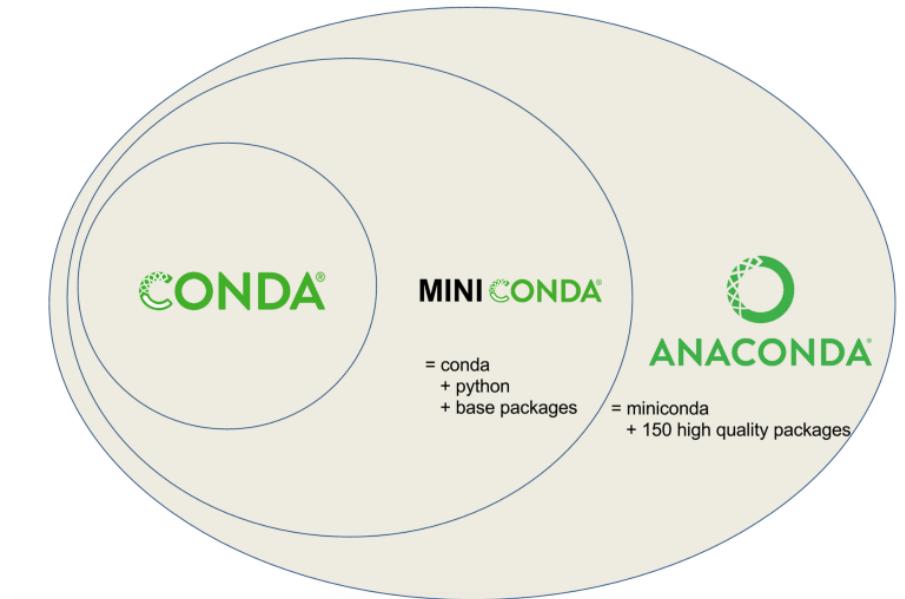
- Now you can import and use this packages in your project

# 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

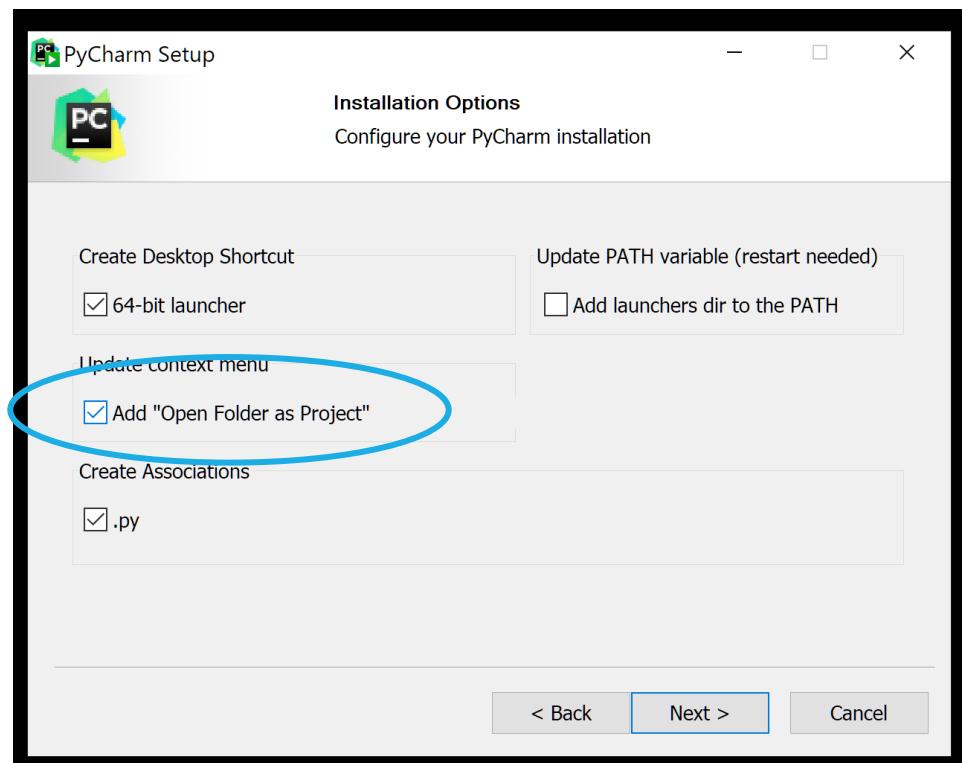
# INSTALLATION 1/2

## 1. Install PyCharm using this links:

- [Linux](#)
- [Windows](#)
- [MacOs](#)

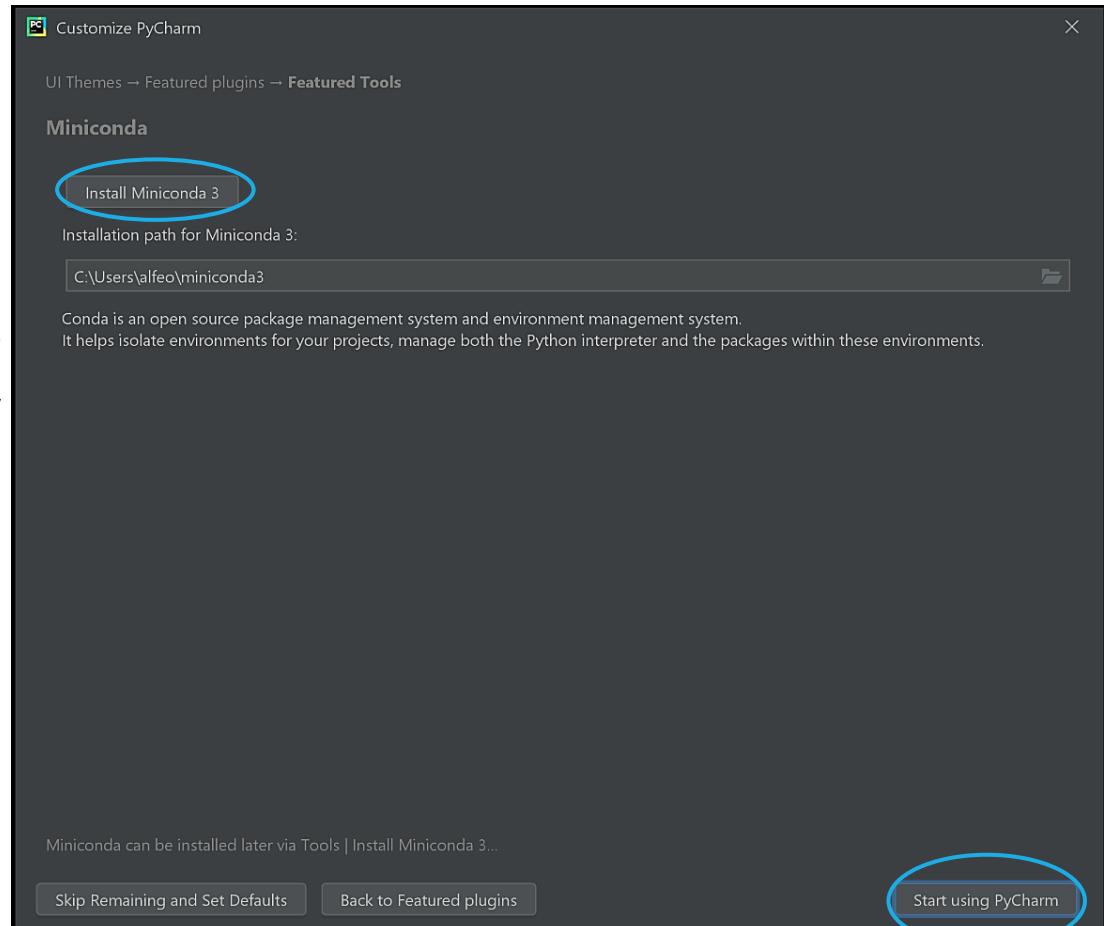
## 2. During PyCharm installation

enable “open folder as project”



# INSTALLATION 2/2

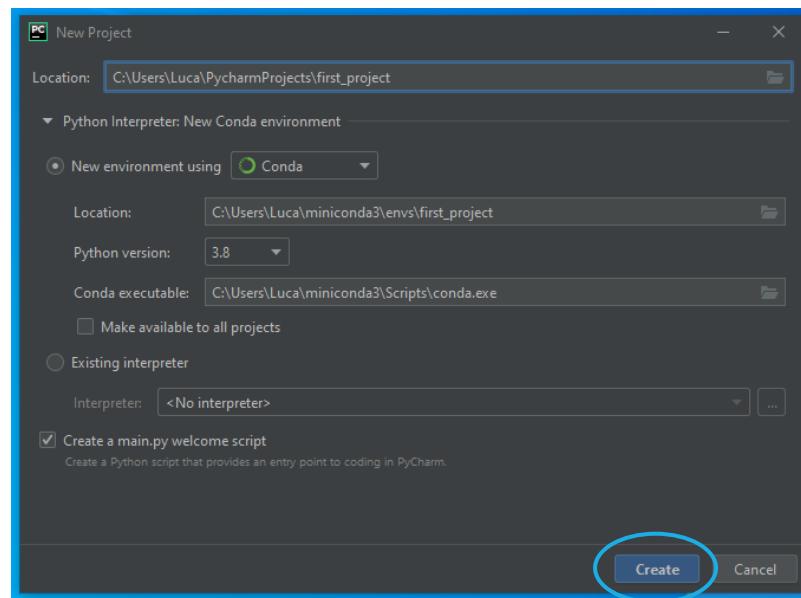
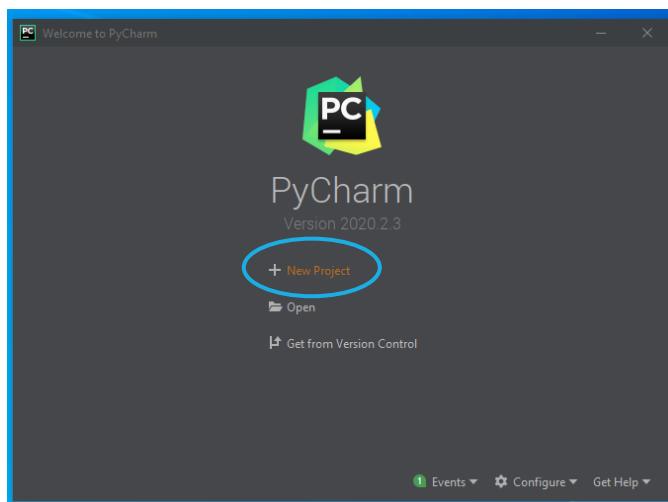
- Accept the JetBrains Privacy Policy
- Choose the UI theme you prefer
- Do not install any featured plugin
- Install Miniconda: includes the conda environment manager, Python, the packages they depend on, and a small number of other useful packages (e.g. pip).
- Remember, Miniconda can be installed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



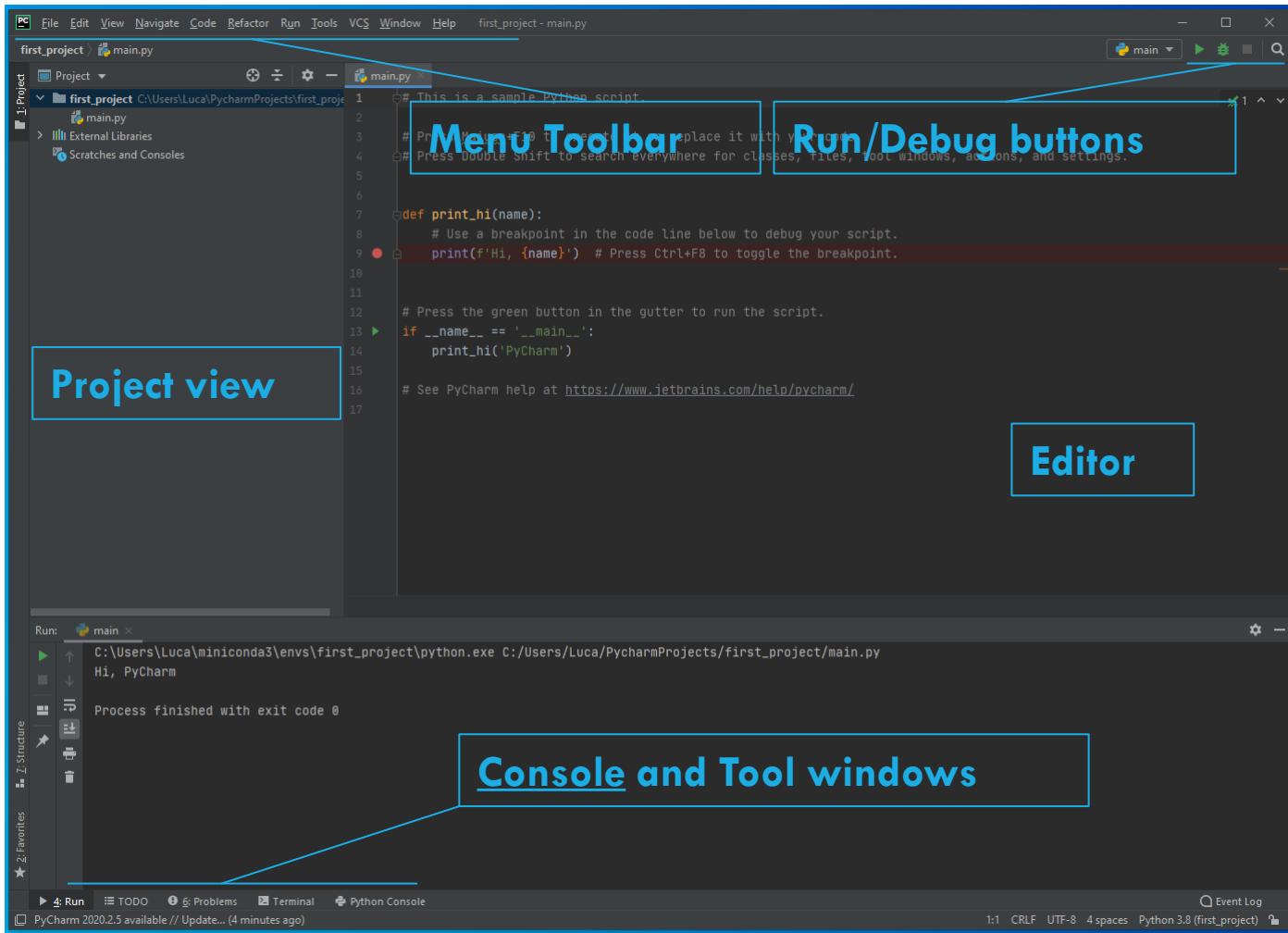
# NEW PROJECT

Create a new project. Name it “first\_project”.

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

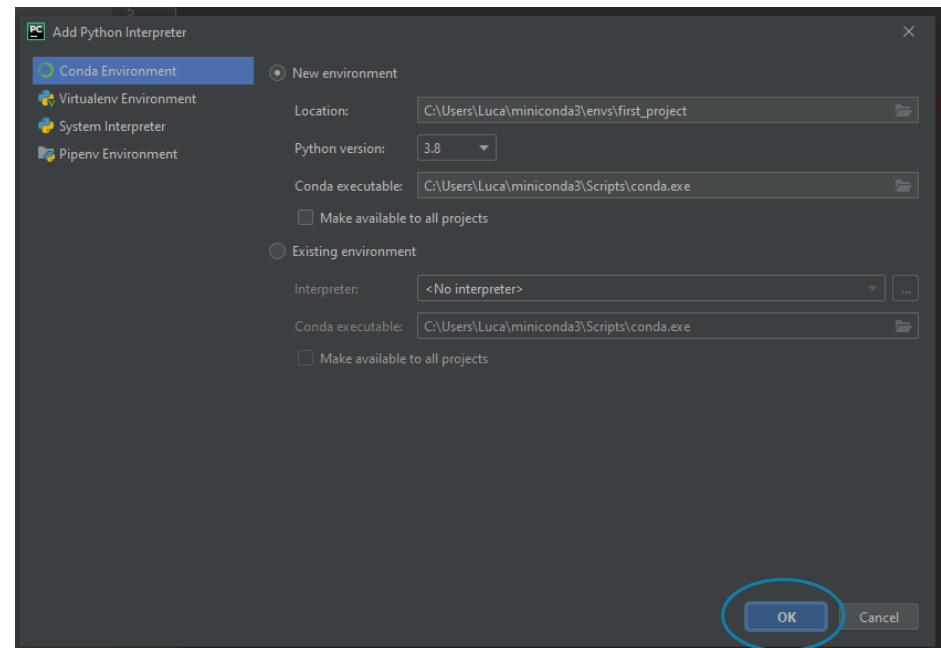
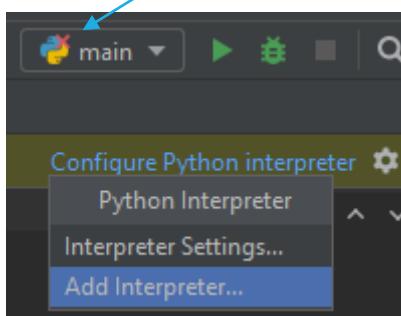


# GUI



# INTERPRETER: DEFAULT CONFIGURATIONS

Is there a red X here?



Now you can run it!



# PYTHON BASICS

```
A dog controller
"""
image = games.load_image("kg.png")
def __init__(self):
    """
    Initialize Dog object and create Text object
    super(Dog, self).__init__(image = Dog.image,
                             x = games.mouse.x,
                             bottom = games.screen.height)
    self.score = games.Text(value = 0, size = 24,
                           top = 5, right = 500)
    add(self.score)
    value = 0.5, game = self)
```

# 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!

The screenshot shows a PyCharm interface with the following code in `main.py`:

```
def print_hi(N, name):
    if N == 1:
        print(f'Hi, my first name is {name}')
    elif N == 2:
        print(f'Hi, my second name is {name}')
    else:
        print(f'Hi, {name}')

if __name__ == '__main__':
    print_hi(1, 'PyCharm')
```

The code uses an `if-elif-else` structure to print different messages based on the value of `N`. It also includes a check for the `__name__` variable to determine if it's being run as the main module.

The terminal output shows the execution of the script and its output:

```
C:\Users\Luca\miniconda3\envs\pythonProject\python.exe C:/Users/Luca/PycharmProjects/pythonProject/main.py
Hi, my first name is PyCharm

Process finished with exit code 0
```

# 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.

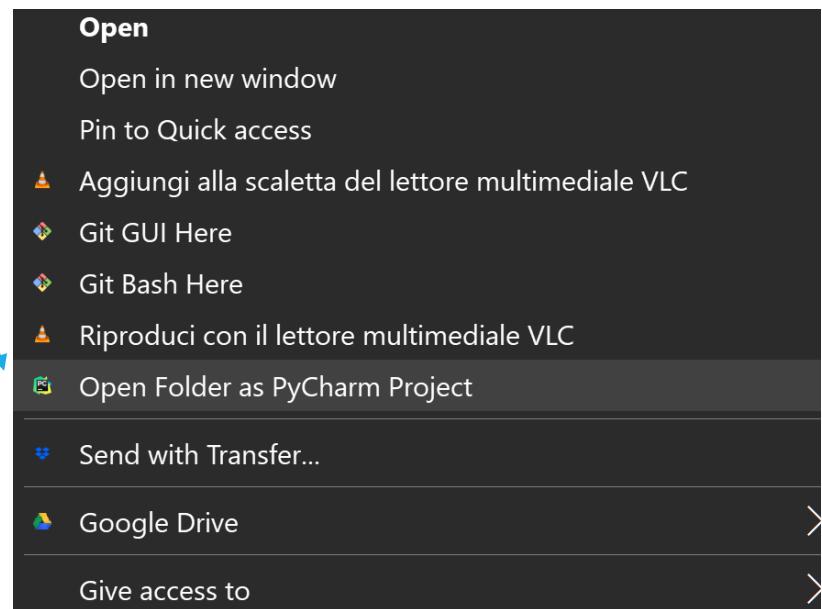
Navigate the project with specialized project views, use shortcuts, jump between files, classes, methods and usages.

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...

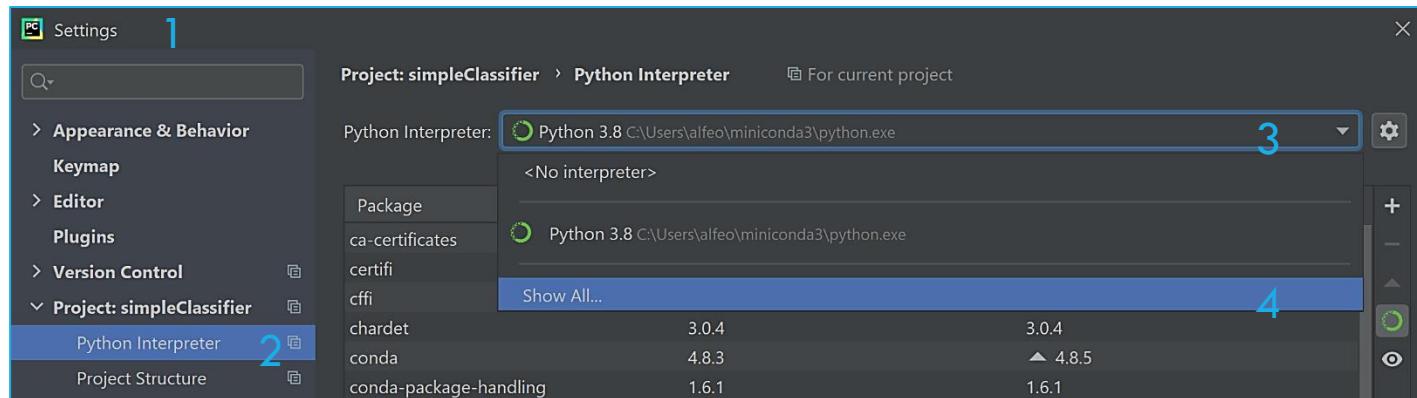
# OPEN A PROJECT

1. Unzip “[simpleClassifier.rar](#)”
2. Open the resulting folder “as a PyCharm project”



# ALTERNATIVE INTERPRETER CONFIGURATION

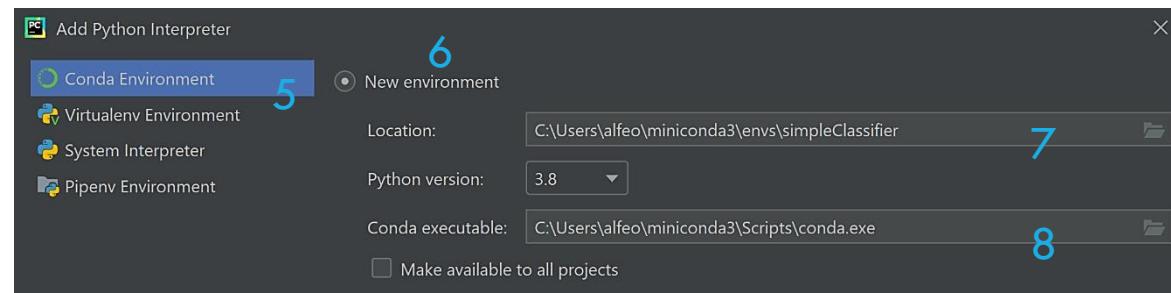
File > Settings > Project > Python interpreter > Show all



+ > Conda Environment > New environment

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

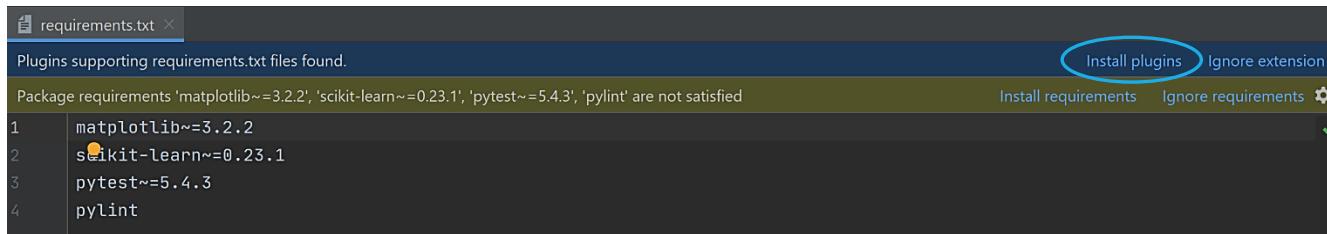
Apply > Ok



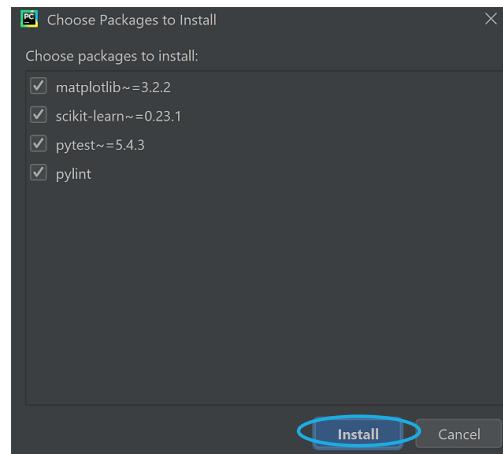
# PROJECT REQUIREMENTS 1/2

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

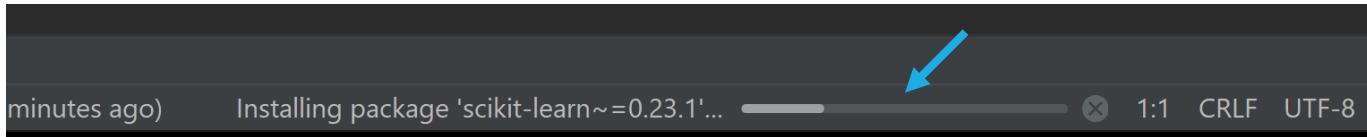


Once the plugin is installed click on “Install requirements”, select all and click install.

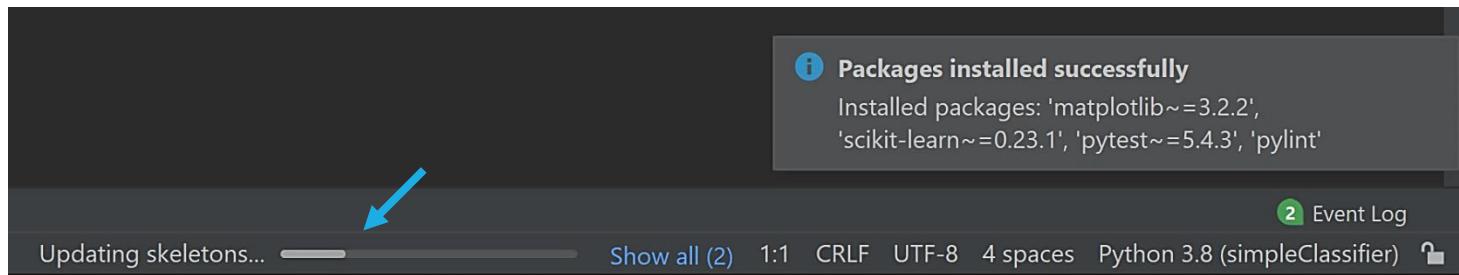


# PROJECT REQUIREMENTS 2/2

- 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".



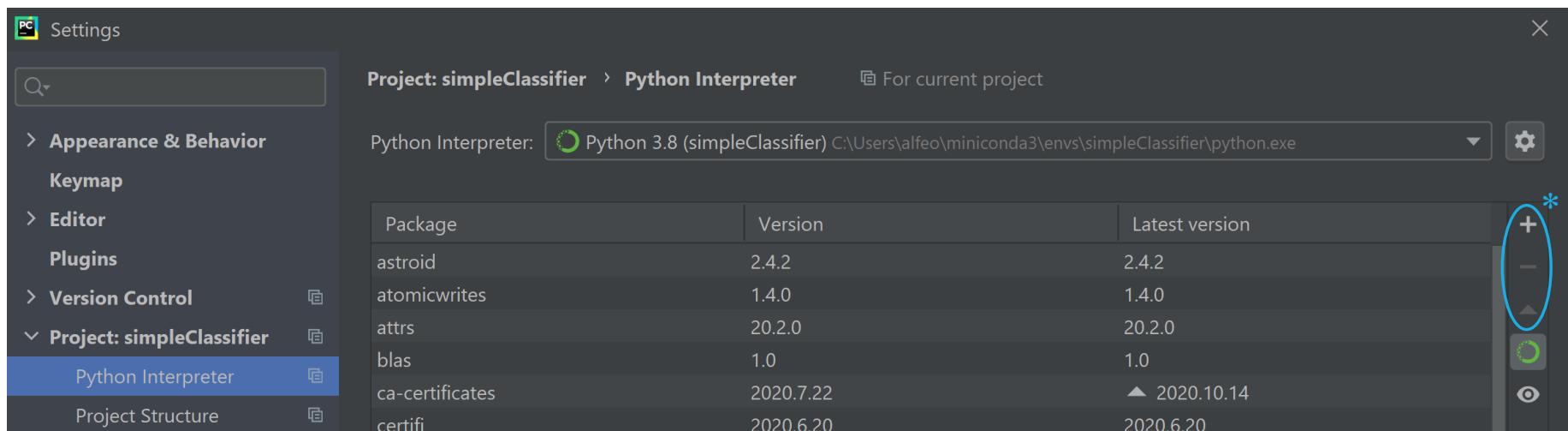
- Each package can also be installed via GUI or with the Terminal by using
  - “`pip install <name_lib>`” to install a single package
  - “`pip install -r requirements.txt`” to install the packages on the requirements.txt

# 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!

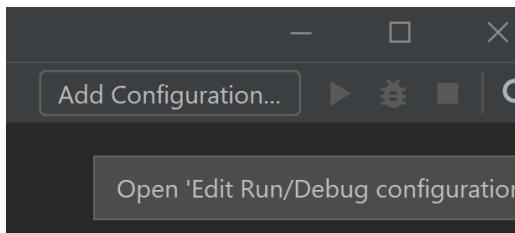
Once done you can see ([add](#), [eliminate](#) and [upgrade\\*](#)) the packages and libraries available in your virtual environment by checking :

File > Settings > Project > Python Interpreter

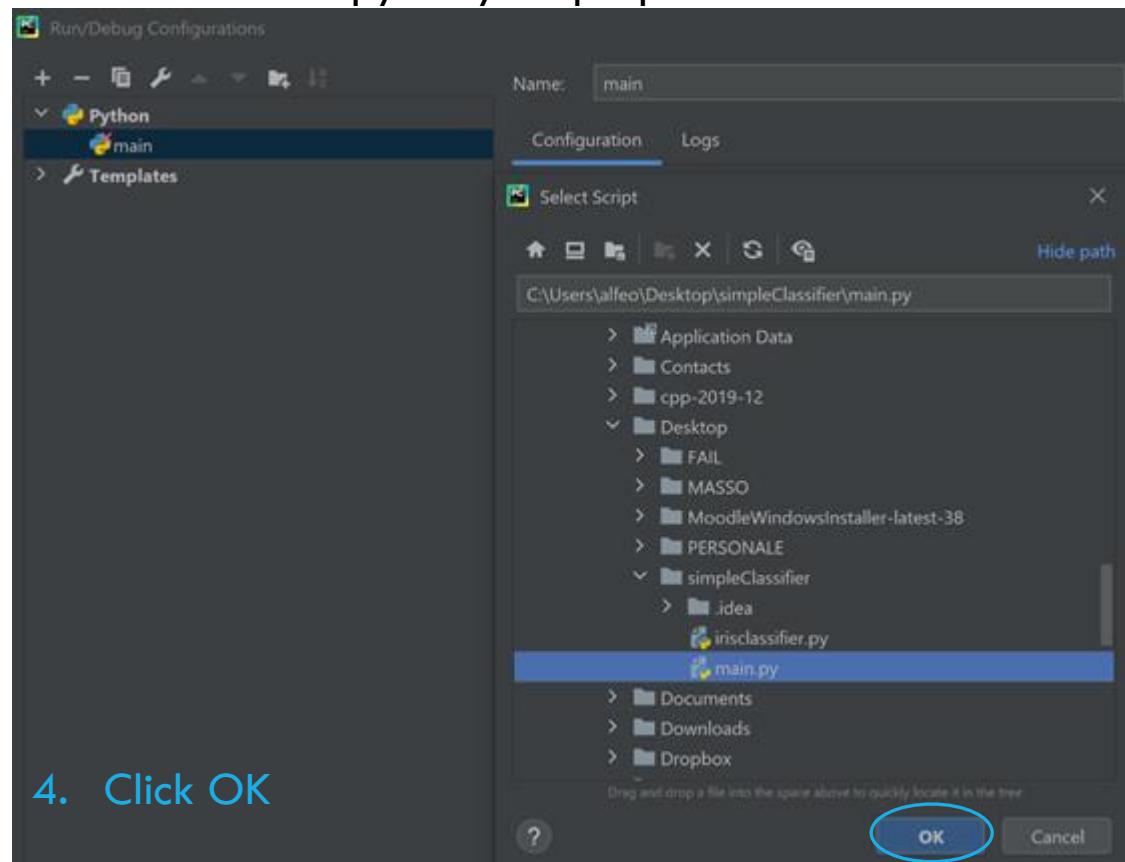


# CONFIGURE THE FIRST RUN

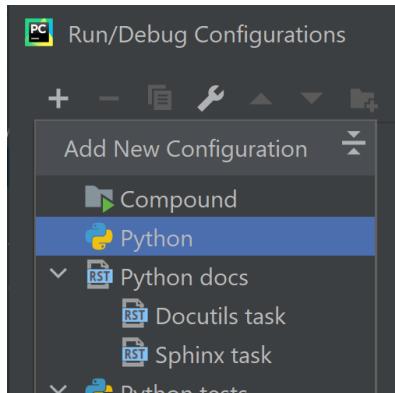
1. Click “Add configuration”



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



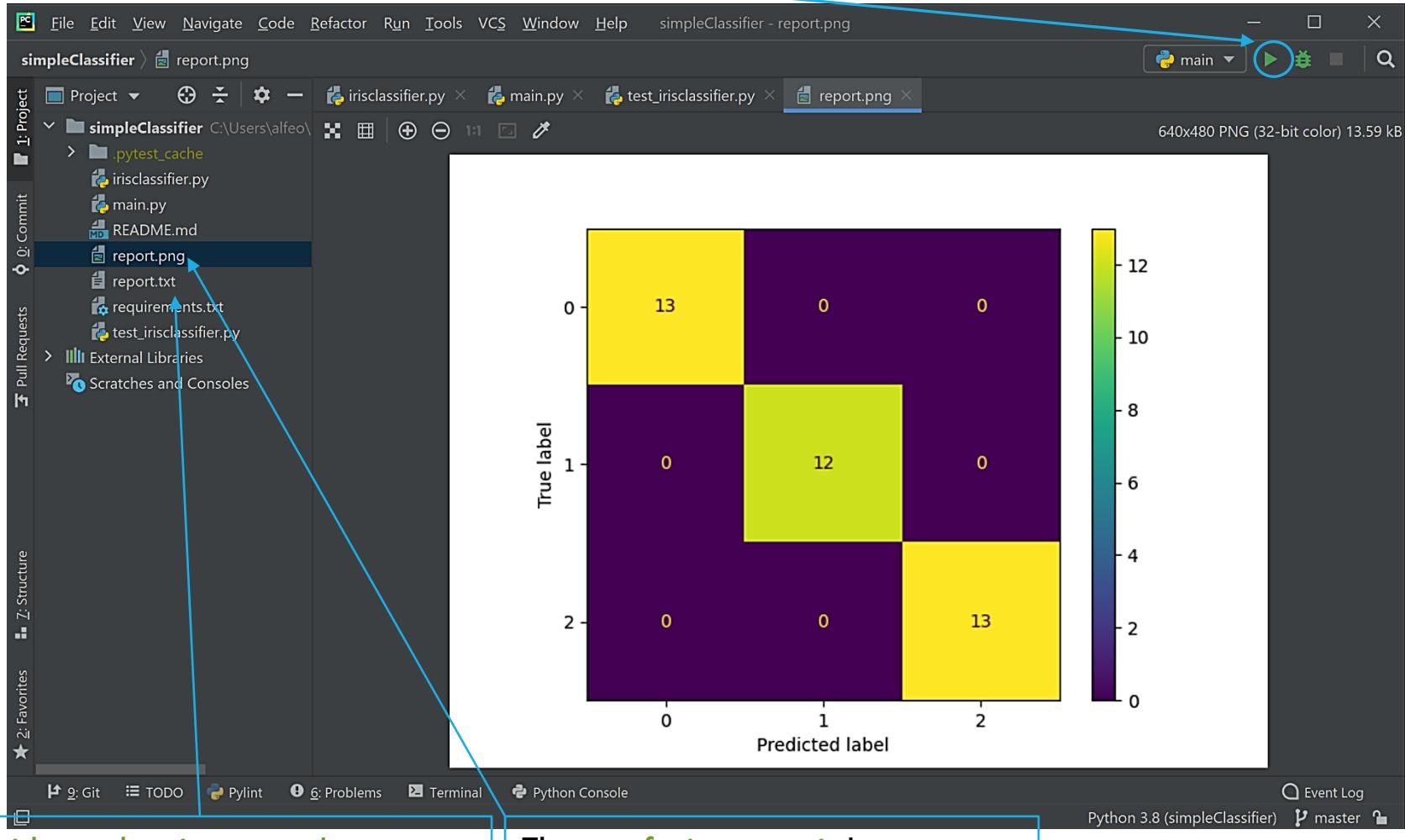
2. Select “Python”



4. Click OK

# RUN IT!

Click RUN



Txt with evaluation score!

The confusion matrix!

# INTRODUCTION TO PACKAGES FOR YOUR PROJECT



Based on previous lecture by A. L. Alfeo

# DATA ANALYSIS IN PYTHON

So many packages available:

- Pylint: to produce readable code
- Pandas: read data from a broad range of sources like CSV, and Excel files.
- Json and Joblib: to save/read configurations and models to/from files.
- Sci-kit learn: machine learning library with a broad range of classification algorithms. It provides also data pre-processing, pipelining and performance analysis modules.
- Matplotlib: graphic visualization libraries.

...and much more.

# 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

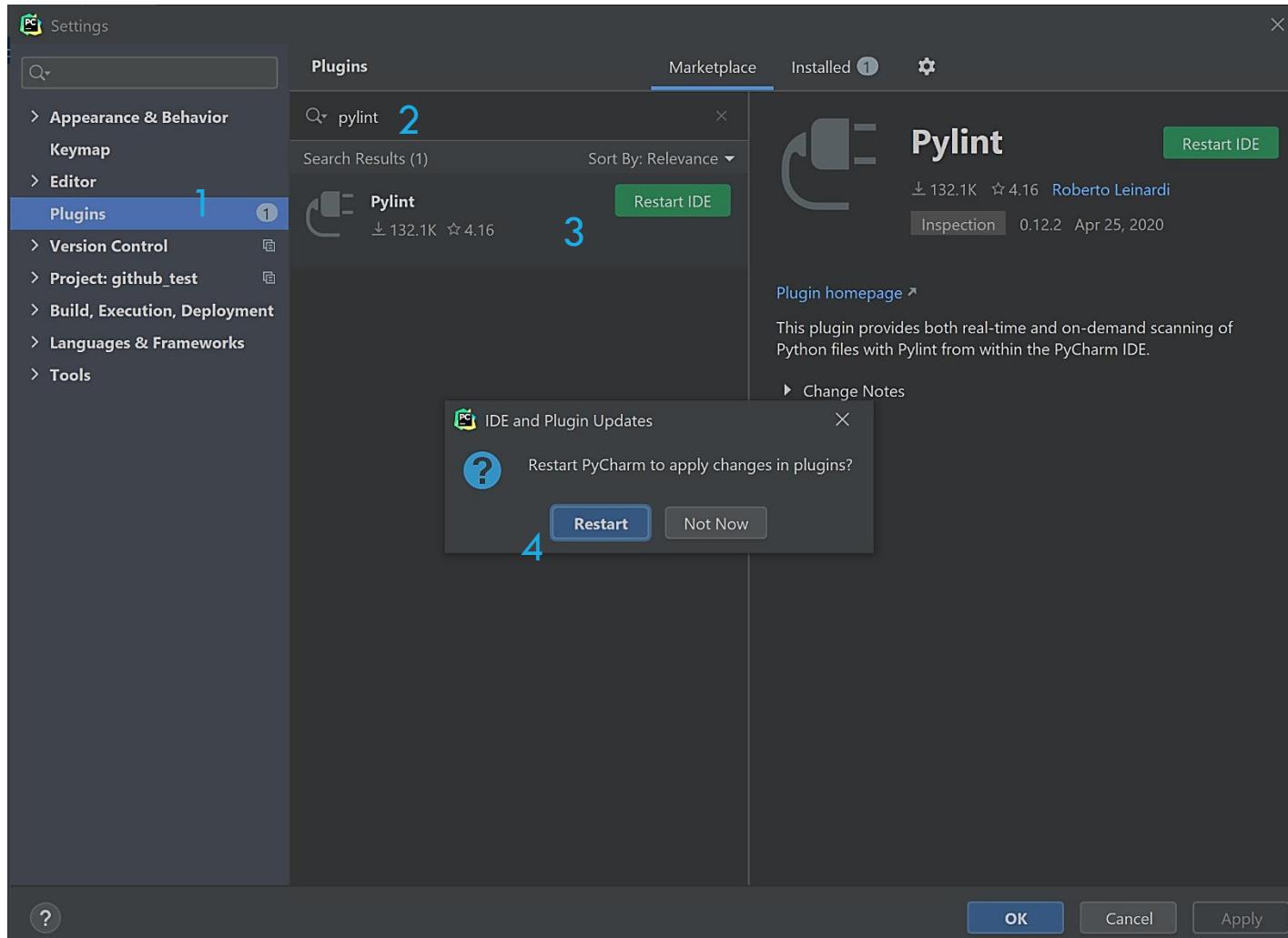
- if declared interfaces are truly implemented

- 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



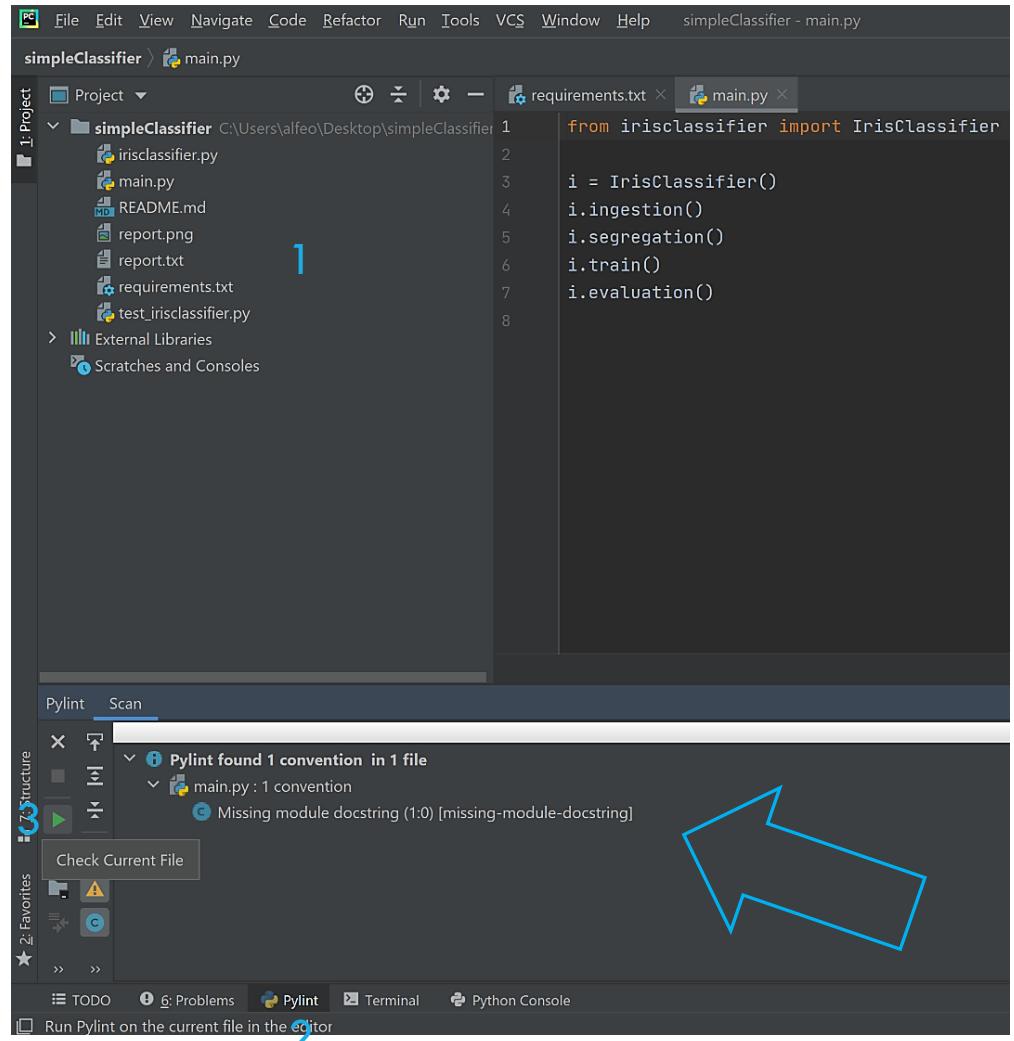
# PYLINT PLUGIN SETUP

The screenshot shows the PyCharm settings interface. On the left, the 'Settings' sidebar is open, with the 'Pylint' tab selected (marked with a blue border and the number 1). The main area displays the 'Pylint' configuration screen. At the top, it says 'For current project'. Below that, there are three fields: 'Path to Pylint executable' containing 'C:\Users\alfeo\anaconda3\envs\github\_test\Scripts\pylint.exe' (marked with the number 2), 'Path to pylintrc:' (marked with the number 3), and 'Arguments:'. At the bottom right of the dialog are 'OK', 'Cancel', and 'Apply' buttons. A status message at the bottom right says 'Success: executable found!'.

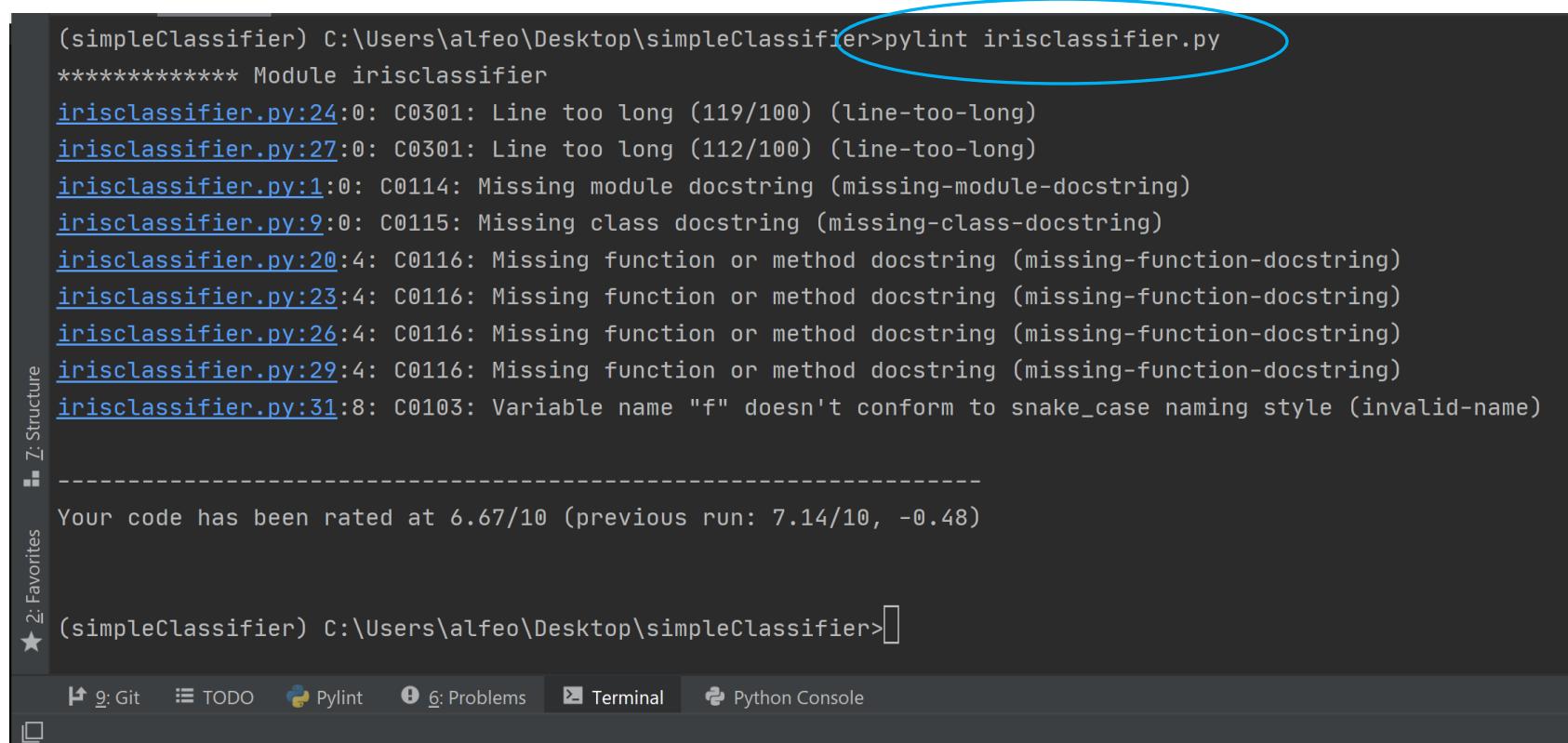
1. File > Settings > Pylint
2. Insert the path to the pylint executable
3. Click “Test”
4. Check the result
5. Click “Apply” and “Ok”. You can now run “pylint” by clicking the green arrow in the Pylint Tool Windows

# 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



```
(simpleClassifier) C:\Users\alfeo\Desktop\simpleClassifier>pylint irisclassifier.py
*****
Module irisclassifier
irisclassifier.py:24:0: C0301: Line too long (119/100) (line-too-long)
irisclassifier.py:27:0: C0301: Line too long (112/100) (line-too-long)
irisclassifier.py:1:0: C0114: Missing module docstring (missing-module-docstring)
irisclassifier.py:9:0: C0115: Missing class docstring (missing-class-docstring)
irisclassifier.py:20:4: C0116: Missing function or method docstring (missing-function-docstring)
irisclassifier.py:23:4: C0116: Missing function or method docstring (missing-function-docstring)
irisclassifier.py:26:4: C0116: Missing function or method docstring (missing-function-docstring)
irisclassifier.py:29:4: C0116: Missing function or method docstring (missing-function-docstring)
irisclassifier.py:31:8: C0103: Variable name "f" doesn't conform to snake_case naming style (invalid-name)

-----
Your code has been rated at 6.67/10 (previous run: 7.14/10, -0.48)

(simpleClassifier) C:\Users\alfeo\Desktop\simpleClassifier>
```

UI elements on the left:

- Z: Structure
- 
- 2: Favorites

Bottom navigation bar:

- Git
- TODO
- Pylint
- Problems
- Terminal
- Python Console

# MORE ON CODING ASSISTANCE

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

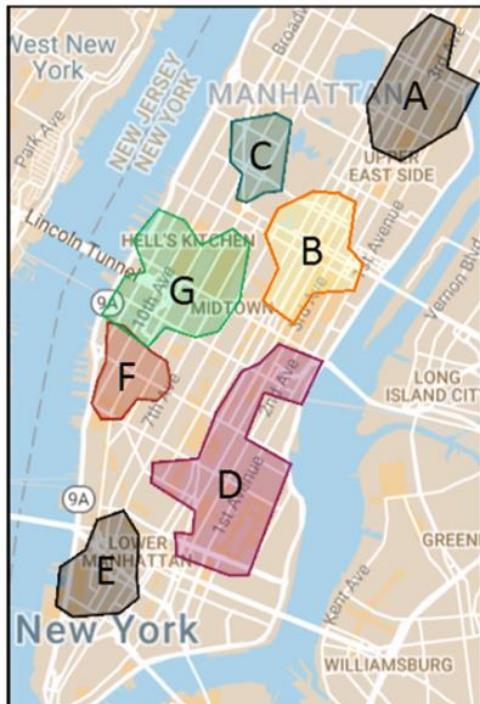
- Python refactoring:
  - renaming
  - extract methods
  - introduce variables and constants
- Coding assistance and analysis:
  - code completion
  - syntax and error highlighting
  - quick fixes
  - guided import
  - integrated Python debugger
  - code testing and code coverage

# WORKSPACE FOLDER ORGANIZATION

The project folder and subfolder organization is up to you, but needs to be simple and clear, and allow to understand the processing put in place by each module.

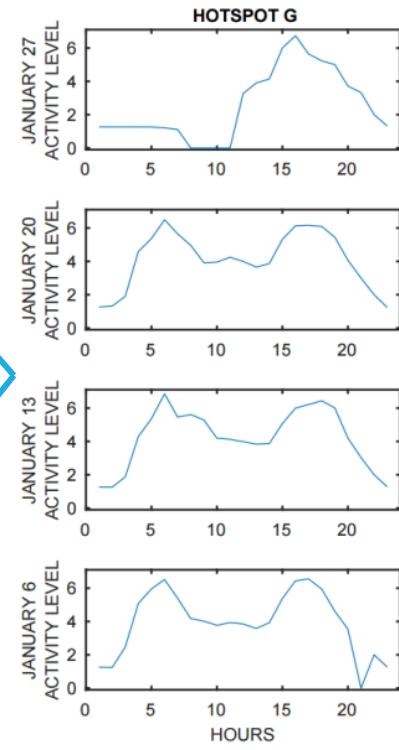
# AN EXAMPLE...

Let's start with a very simple example ([click to download the data](#)).



HOTSPOT SET

By looking at mobility dynamics in the hotspot it's easy to distinguish between weekdays and weekend, or even to detect traffic anomalies (i.e. an parade, a storm).



# UPLOAD/SAVE TABULAR DATA WITH PANDAS.DATFRAME

```
from pandas import read_csv

# Read csv
data1 = read_csv('data/hotspotUrbanMobility-1.csv')
# Print dataframe shape
print(data1.shape) ← Path of the csv to read

# Read csv
data2 = read_csv('data/hotspotUrbanMobility-2.csv')
# Print dataframe shape
print(data2.shape)
# Append dataframes
data1 = data1.append(data2, ignore_index=True) ← Dataframe shape consist of its numer of rows and column
print(data1.shape)

# Save the new datafame as csv
data1.to_csv('data/completeDataset.csv', index=False) ← 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
```

# CREATING DATABASES WITH SQLITE3

SQLite is a database engine that makes it simple to store and work with relational data. In Python it can be used via `sqlite3`!

```
import sqlite3

con = sqlite3.connect('example.db')
cur = con.cursor()
# Create table and insert a row of data
cur.execute('CREATE TABLE <table_name> (<col_1>, <col_2>, <...>)')
cur.execute('INSERT INTO <table_name> VALUES ((<val_1>, <val_2>, <...>))')
# Save the changes
con.commit()
# Return all results of query
cur.execute('SELECT <...> FROM <...> WHERE <...> ')
cur.fetchall()
# Return first result of query
cur.execute('SELECT <...> FROM <...> WHERE <...> ')
cur.fetchone()
# Close the connection
con.close()
```

# PANDAS FOR SQLITE3 DATABASES

Once you have a working connection to the SQL database, you can use read\_sql to query it or obtain a dataframe out of it and to\_sql to save a dataframe as an SQL table.

```
import sqlite3
import pandas as pd
from sklearn.datasets import load_iris
from pandas import read_csv

con = sqlite3.connect('example.db')
# Obtain a dataframe
data = read_csv('data/preprocessedDataset.csv')
# Save it to the database
data.to_sql("mobility", con, if_exists="replace")
# Check the database
print(pd.read_sql("select * from mobility;", con))
# Save the changes and close the connection
con.commit()
con.close()
```

# TRY THE LAST 2 SLIDES YOURSELF!

```
      index  Anomalous  Cluster  Day  ...      h20      h21      h22      h23
0          0         0       1    2  ...  1.000000  1.000000  1.000000  0.938944
1          1         0       1    3  ...  1.000000  1.000000  0.998011  0.794555
2          2         0       2    4  ...  0.610982  0.000000  0.252500  0.206939
3          3         0       0    5  ...  0.464342  0.317294  0.139459  0.000000
4          4         0       0    6  ...  0.772373  0.461212  0.000000  0.226510
...
323      323         0       0   23  ...  0.742926  0.530848  0.484248  0.000000
324      324         0       2   27  ...  0.730543  0.494042  0.000000  0.287482
325      325         0       0   28  ...  0.599699  0.510087  0.578106  0.289053
326      326         0       0   29  ...  0.989600  0.774025  0.523705  0.484244
327      327         0       0   30  ...  1.000000  0.853960  0.563657  0.494982
```

[328 rows x 28 columns]

Process finished with exit code 0

# MORE ON TABULAR DATA WITH PANDAS

- Normalize JSON and convert them into dataframe
- 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

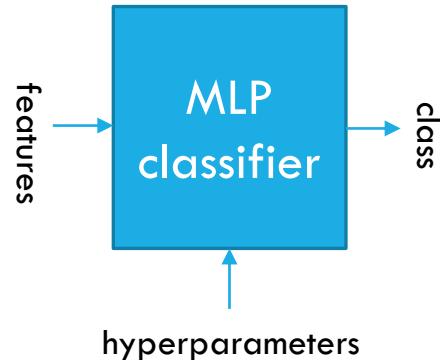
Scikit-learn provides an easy-to-use **classifier** based on artificial neural networks. Those use features, information extracted from an input data instance, to classify the instance itself. [Here you can find the documentation to use them.](#)

```
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**
- **Batch size**
- **Max iterations**



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

# CLASSIFIER 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)))
```

The diagram illustrates the workflow of a machine learning pipeline using Joblib. It consists of four main steps:

- Random\_state for reproducibility**: Points to the line `initial_model = MLPClassifier(random_state=0).fit(training_data, ravel(training_labels))`.
- Save a fitted model to file**: Points to the line `joblib.dump(initial_model, 'config/fitted_model.sav')`.
- Load a fitted model from file**: Points to the line `model = joblib.load('config/fitted_model.sav')`.
- Use the model with new data**: Points to the line `print(model.score(testing_data, ravel(testing_labels)))`.

# 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])
# ...try to use the default 70-30 split
```

```
mlp = MLPClassifier(random_state=0)
parameters = {'max_iter': (100, 200, 300)}
# ...try to use all the hyper-params at slide 63
gs = GridSearchCV(mlp, parameters)
#...try PredefinedSplit...
gs.fit(training_data, ravel(training_labels))
#...or try to pick the first runner up config
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 classifier?

# MORE ON MODEL EVALUATION...

- A number of [algorithms](#) and performance [metrics](#) available on scikit-learn
- [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.PYTHON](#)

Create beautiful graphs (here you have the [cheatsheets](#) and a few tutorials [1, 2, 3]):

- [Plot](#) and [save a figure](#)
- [Histogram](#) and barplot
- [Boxplots](#)
- [Scatterplot](#)
- Create single and [multiple graphs](#)

# IMPROVE YOUR TEAMWORK!



Based on previous lecture by A. L. Alfeo

# 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** for the functionalities 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)
- Implement **DELETE** and **PUT** functionalities, that's up to you!

# REST API AS INTERFACES 2\3

In Python you can use [Flask](#) and [Flask-Restful](#) to build and use a simple REST server. For instance you can define the end-point «/» as it follows

```
from pandas import read_csv
from flask import Flask, request
from flask_restful import Resource, Api

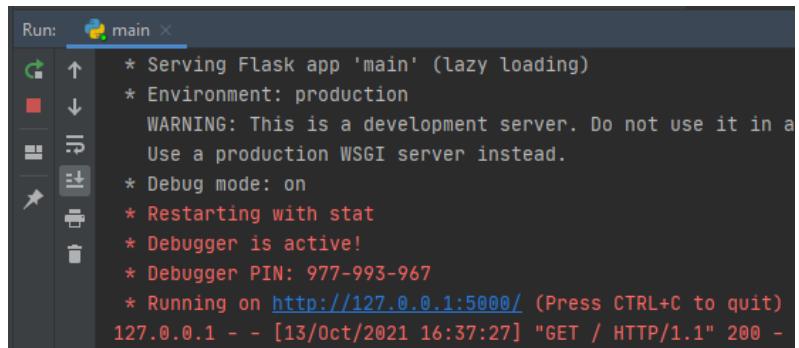
app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        data = read_csv('data/preprocessedDataset.csv')
        data = data.to_dict()
        return {'data': data}, 200

    def post(self):
        some_json = request.get_json()
        return {'you sent ': some_json}, 201

api.add_resource(HelloWorld, '/')

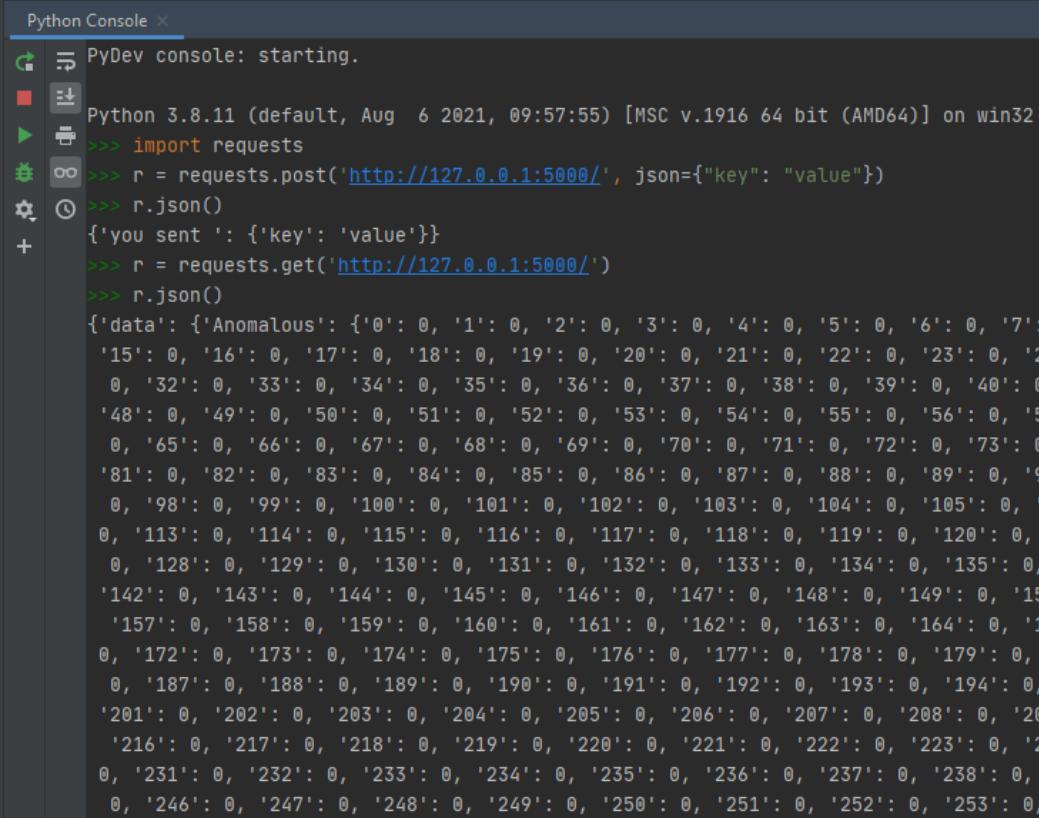
if __name__ == '__main__':
    app.run(debug=True)
```



```
* Serving Flask app 'main' (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a
         Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 977-993-967
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Oct/2021 16:37:27] "GET / HTTP/1.1" 200 -
```

# TRY IT YOURSELF USING REQUEST

Send a request via Python Console



The screenshot shows a Python console window titled "Python Console". The console output is as follows:

```
PyDev console: starting.

Python 3.8.11 (default, Aug  6 2021, 09:57:55) [MSC v.1916 64 bit (AMD64)] on win32
>>> import requests
>>> r = requests.post('http://127.0.0.1:5000/', json={"key": "value"})
>>> r.json()
{'you sent ': {'key': 'value'}}
>>> r = requests.get('http://127.0.0.1:5000/')
>>> r.json()
{'data': {'Anomalous': {'0': 0, '1': 0, '2': 0, '3': 0, '4': 0, '5': 0, '6': 0, '7': 0, '8': 0, '9': 0, '10': 0, '11': 0, '12': 0, '13': 0, '14': 0, '15': 0, '16': 0, '17': 0, '18': 0, '19': 0, '20': 0, '21': 0, '22': 0, '23': 0, '24': 0, '25': 0, '26': 0, '27': 0, '28': 0, '29': 0, '30': 0, '31': 0, '32': 0, '33': 0, '34': 0, '35': 0, '36': 0, '37': 0, '38': 0, '39': 0, '40': 0, '41': 0, '42': 0, '43': 0, '44': 0, '45': 0, '46': 0, '47': 0, '48': 0, '49': 0, '50': 0, '51': 0, '52': 0, '53': 0, '54': 0, '55': 0, '56': 0, '57': 0, '58': 0, '59': 0, '60': 0, '61': 0, '62': 0, '63': 0, '64': 0, '65': 0, '66': 0, '67': 0, '68': 0, '69': 0, '70': 0, '71': 0, '72': 0, '73': 0, '74': 0, '75': 0, '76': 0, '77': 0, '78': 0, '79': 0, '80': 0, '81': 0, '82': 0, '83': 0, '84': 0, '85': 0, '86': 0, '87': 0, '88': 0, '89': 0, '90': 0, '91': 0, '92': 0, '93': 0, '94': 0, '95': 0, '96': 0, '97': 0, '98': 0, '99': 0, '100': 0, '101': 0, '102': 0, '103': 0, '104': 0, '105': 0, '106': 0, '107': 0, '108': 0, '109': 0, '110': 0, '111': 0, '112': 0, '113': 0, '114': 0, '115': 0, '116': 0, '117': 0, '118': 0, '119': 0, '120': 0, '121': 0, '122': 0, '123': 0, '124': 0, '125': 0, '126': 0, '127': 0, '128': 0, '129': 0, '130': 0, '131': 0, '132': 0, '133': 0, '134': 0, '135': 0, '136': 0, '137': 0, '138': 0, '139': 0, '140': 0, '141': 0, '142': 0, '143': 0, '144': 0, '145': 0, '146': 0, '147': 0, '148': 0, '149': 0, '150': 0, '151': 0, '152': 0, '153': 0, '154': 0, '155': 0, '156': 0, '157': 0, '158': 0, '159': 0, '160': 0, '161': 0, '162': 0, '163': 0, '164': 0, '165': 0, '166': 0, '167': 0, '168': 0, '169': 0, '170': 0, '171': 0, '172': 0, '173': 0, '174': 0, '175': 0, '176': 0, '177': 0, '178': 0, '179': 0, '180': 0, '181': 0, '182': 0, '183': 0, '184': 0, '185': 0, '186': 0, '187': 0, '188': 0, '189': 0, '190': 0, '191': 0, '192': 0, '193': 0, '194': 0, '195': 0, '196': 0, '197': 0, '198': 0, '199': 0, '200': 0, '201': 0, '202': 0, '203': 0, '204': 0, '205': 0, '206': 0, '207': 0, '208': 0, '209': 0, '210': 0, '211': 0, '212': 0, '213': 0, '214': 0, '215': 0, '216': 0, '217': 0, '218': 0, '219': 0, '220': 0, '221': 0, '222': 0, '223': 0, '224': 0, '225': 0, '226': 0, '227': 0, '228': 0, '229': 0, '230': 0, '231': 0, '232': 0, '233': 0, '234': 0, '235': 0, '236': 0, '237': 0, '238': 0, '239': 0, '240': 0, '241': 0, '242': 0, '243': 0, '244': 0, '245': 0, '246': 0, '247': 0, '248': 0, '249': 0, '250': 0, '251': 0, '252': 0, '253': 0, '254': 0, '255': 0, '256': 0, '257': 0, '258': 0, '259': 0, '260': 0, '261': 0, '262': 0, '263': 0, '264': 0, '265': 0, '266': 0, '267': 0, '268': 0, '269': 0, '270': 0, '271': 0, '272': 0, '273': 0, '274': 0, '275': 0, '276': 0, '277': 0, '278': 0, '279': 0, '280': 0, '281': 0, '282': 0, '283': 0, '284': 0, '285': 0, '286': 0, '287': 0, '288': 0, '289': 0, '290': 0, '291': 0, '292': 0, '293': 0, '294': 0, '295': 0, '296': 0, '297': 0, '298': 0, '299': 0, '2000': 0}}
```

# REST API AS INTERFACES 3\3

You can also use an **argument** with your request as it follows

```
from flask import Flask, request
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class Multi(Resource):
    def get(self, num):
        return {'result': num * 10}

api.add_resource(Multi, '/multi/<int:num>')

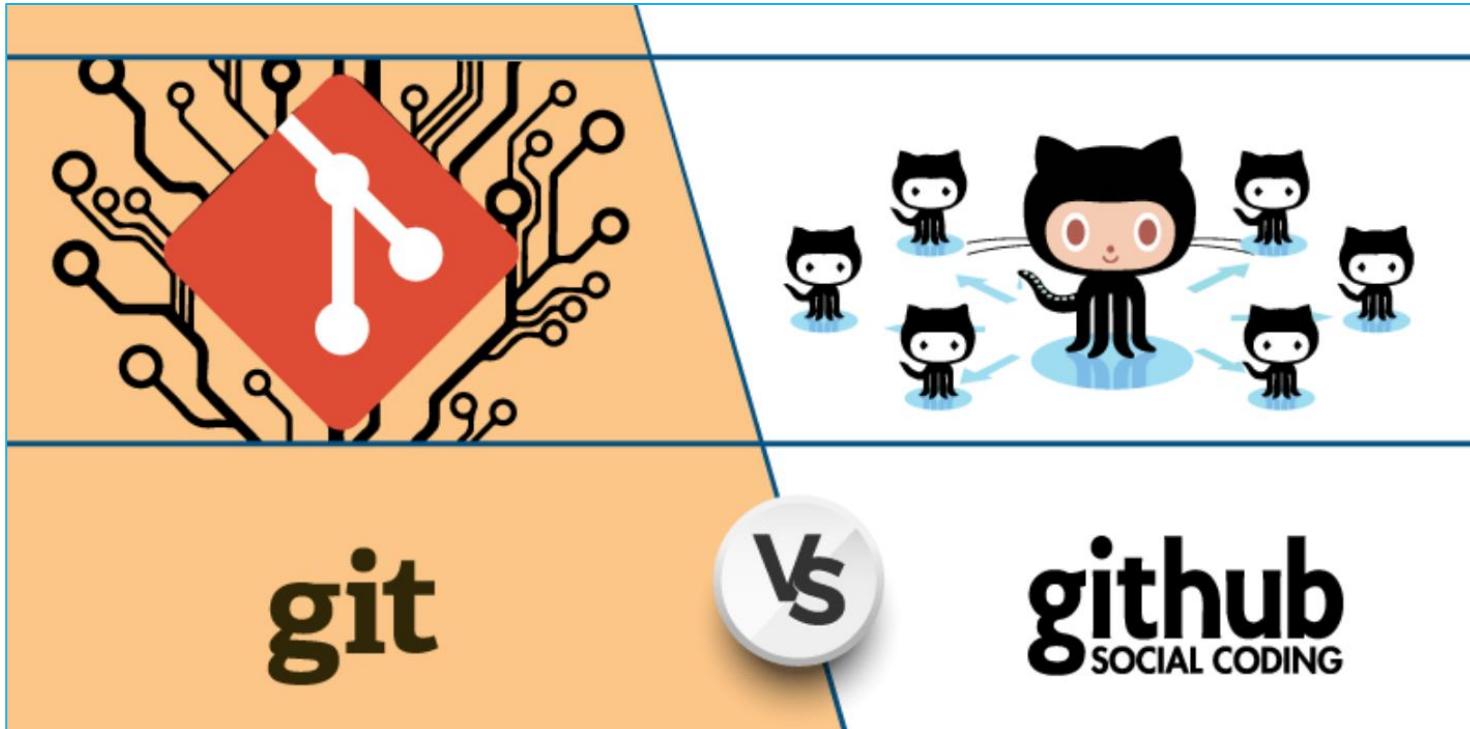
if __name__ == '__main__':
    app.run(debug=True)
```

# TRY IT YOURSELF USING REQUEST

Send a request via Python Console

```
>>> r = requests.get('http://127.0.0.1:5000/multi/4')
>>> r.json()
{'result': 40}
```

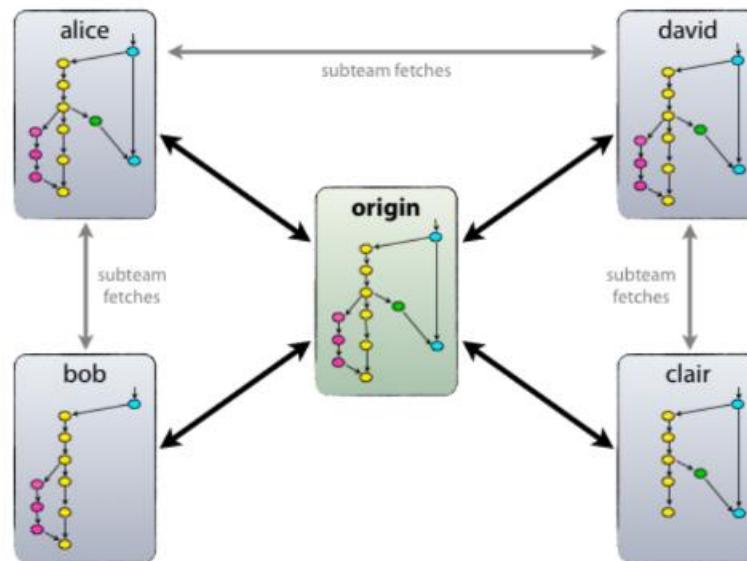
# GIT AND GITHUB



Partially based on the lecture of [F. Galatolo](#)

# GIT

Git is a **distributed Version Control System** in which each participant 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!*)

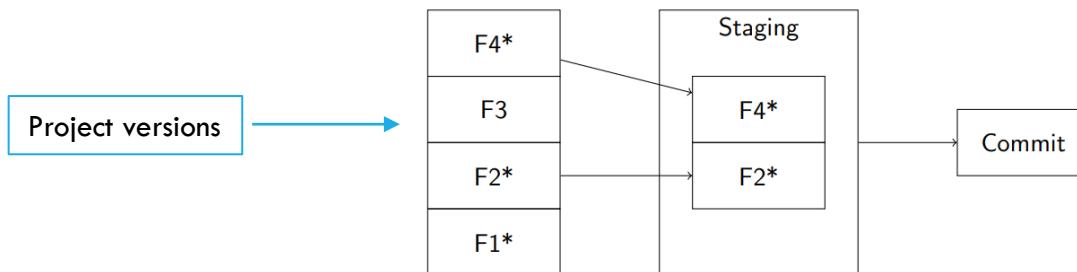
**.gitignore** file containing ignoring rules

**.gitmodules** file containing submodules information

# COMMIT

A **commit** is immutable and atomic units of modification within a project, a snapshot of the project in a given time, it is uniquely identified by its hash and can be commented by a message. A git commit is a two-stages process.

- Add files to the staging environment `git add file/folder`
- Commit the changes `git commit -m "commit message"`



**HEAD** is a Git variable that points to the most recent commit. HEAD can be changed as it follows, the second option change also the files in your repo:

`git reset <NEW_HEAD>`

`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>`

# ... AND MERGE

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

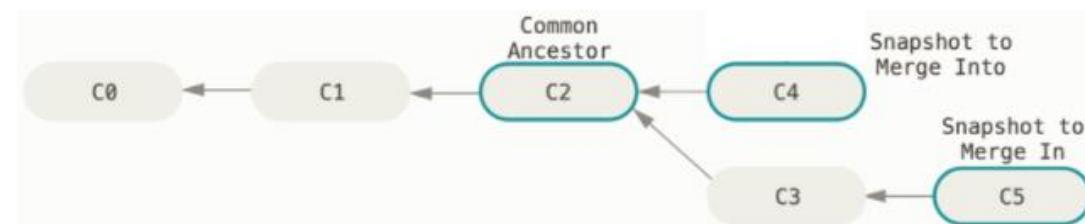
All you have to do is **check out** the branch you wish to merge into and then run the `git merge` command:

## • `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**, as it may happen in your project since each one of you should work on a different module!

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>**

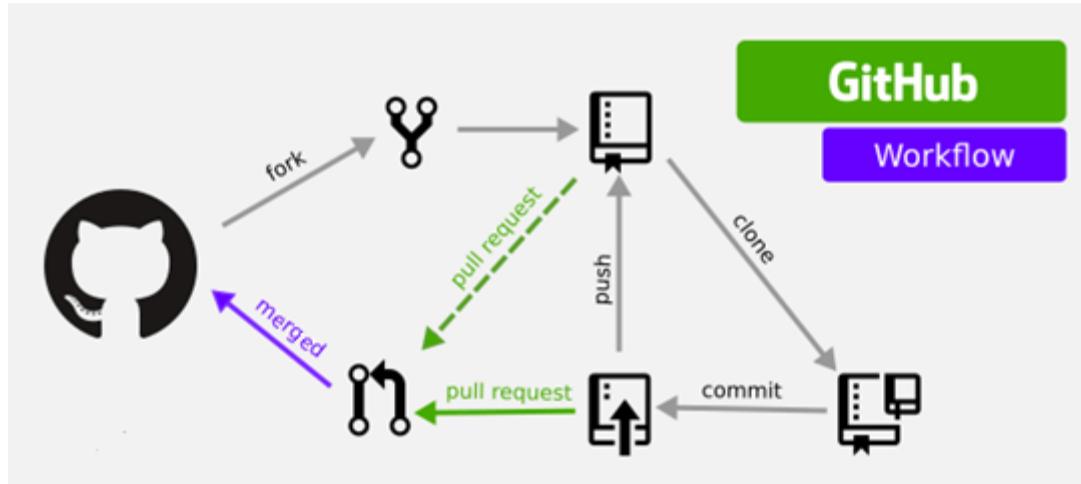
- Push the state of the local branch to the origin branch <branch>

# 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

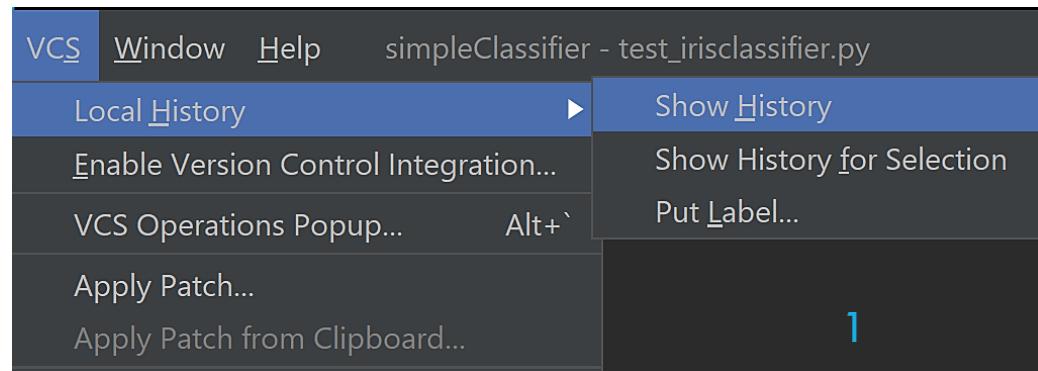


Based on previous lecture by A. L. Alfeo

# 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

A screenshot of PyCharm's "Side-by-side viewer" tool, which displays a code diff between two versions of a file. The left pane shows the initial state of the code, and the right pane shows the current state. Changes are highlighted with green and red annotations. A large blue number "2" is overlaid on the left side of the interface. The status bar at the bottom indicates "1 difference".

```
from typing import Set
import irisclassifier
import pytest
```

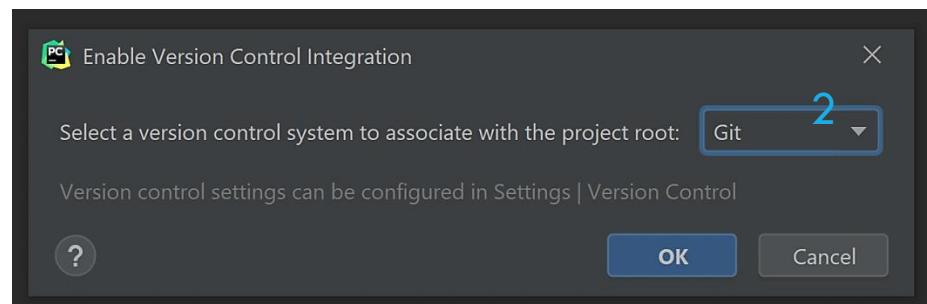
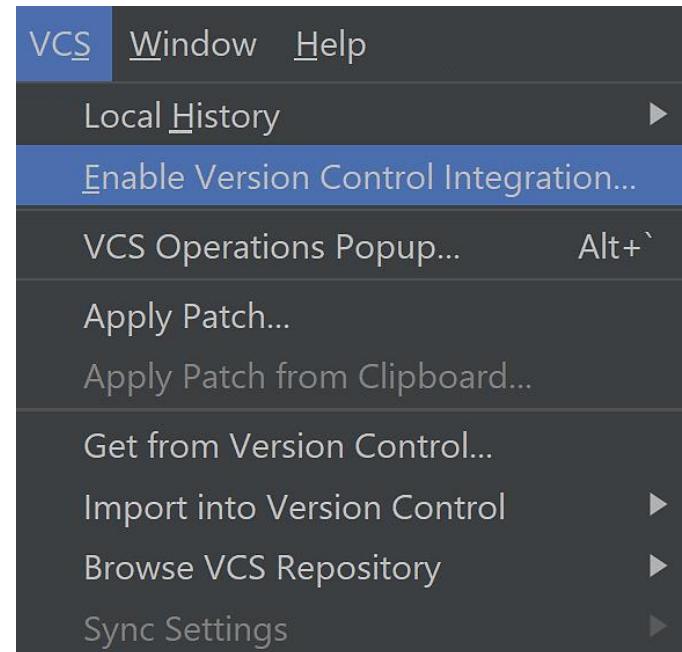
```
from typing import Set
import irisclassifier
import pytest
```

```
def test_evaluation():
```

# MANAGE PROJECTS WITH GITHUB

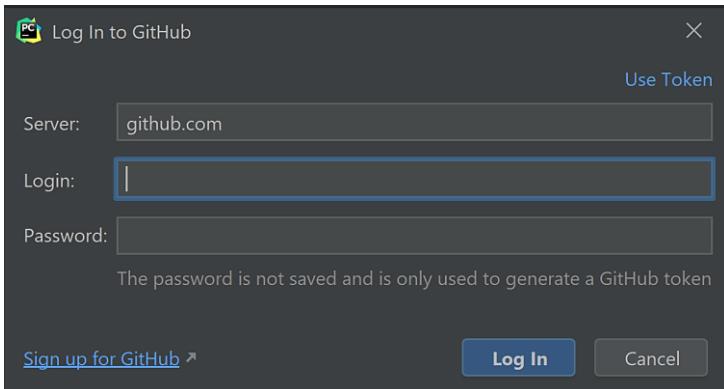
**Github** is way more convenient especially when a project is developed in group. Now you need to:

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.

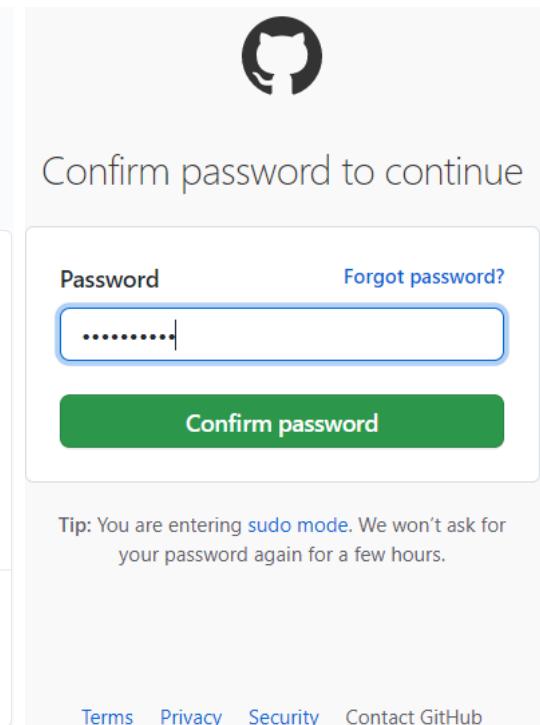
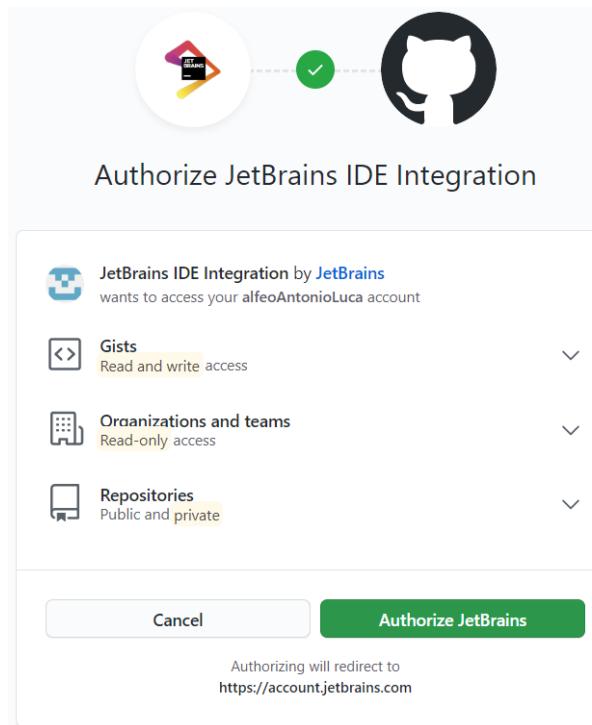


# ENABLE VCI TO USE GITHUB

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

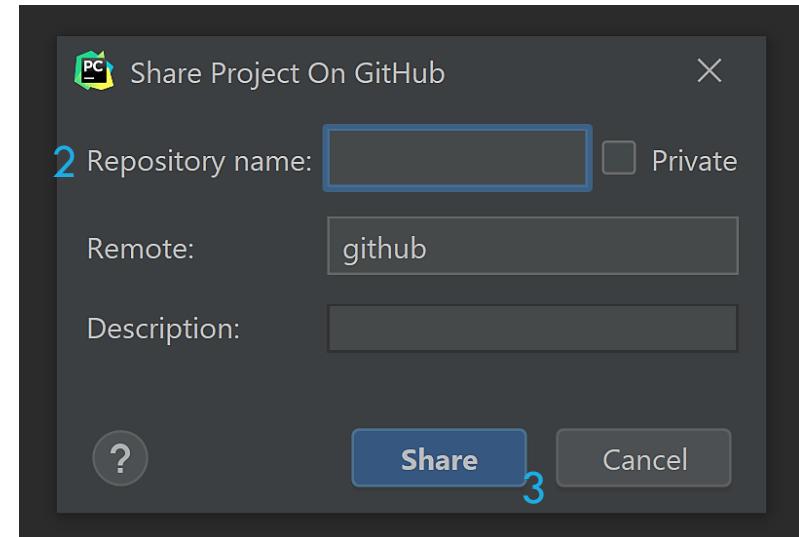
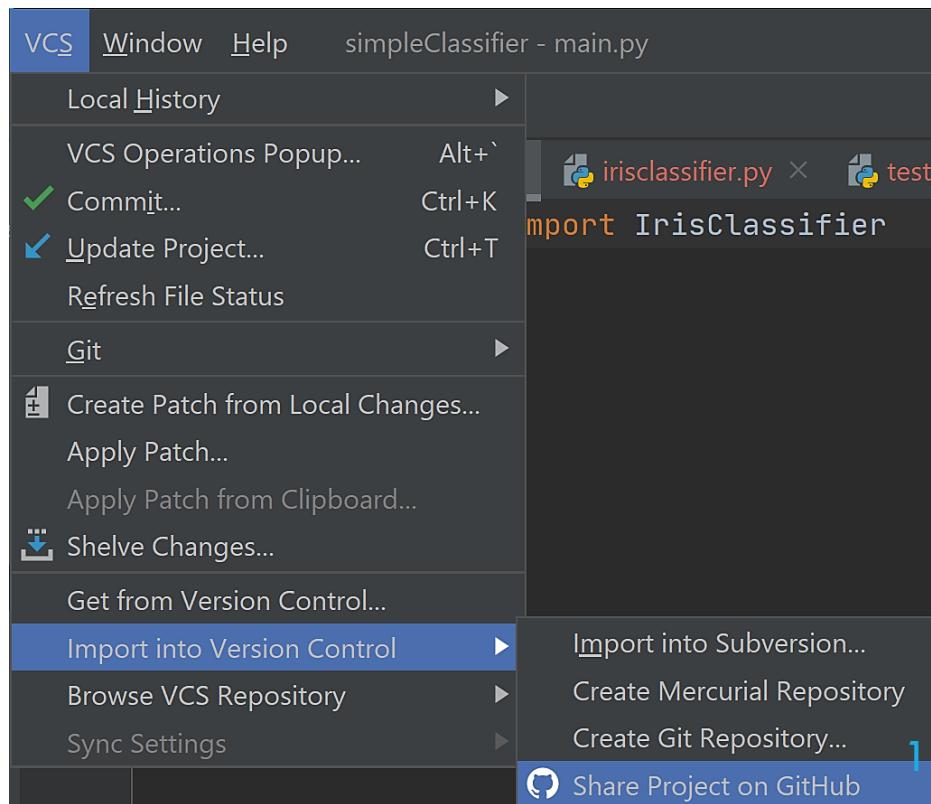


A screenshot of a "Log In to GitHub" dialog box. It has fields for "Server" (github.com), "Login" (empty), and "Password" (empty). Below the password field is a note: "The password is not saved and is only used to generate a GitHub token". At the bottom are "Sign up for GitHub" and "Log In" buttons.



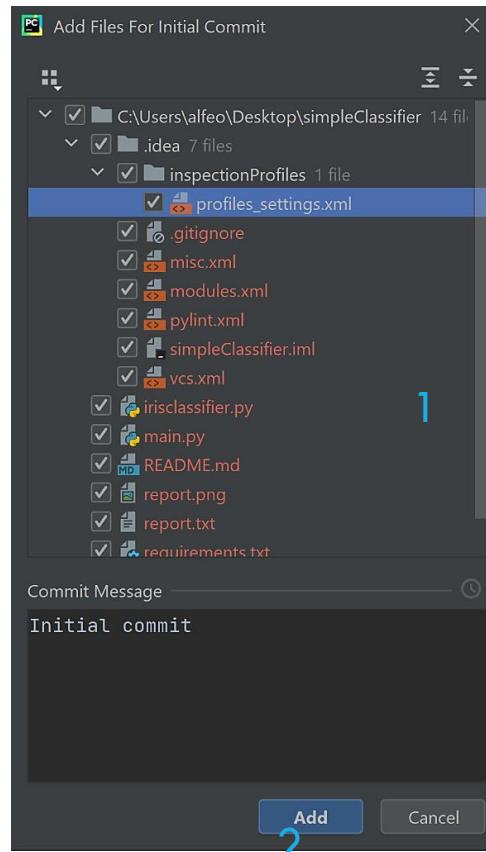
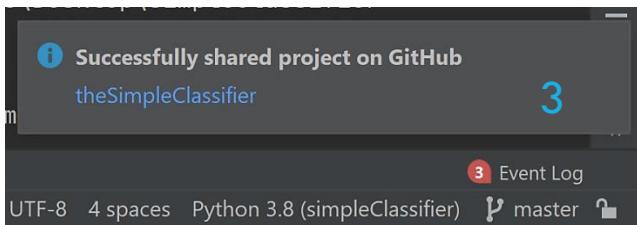
# SHARE A PROJECT 1/2

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

The screenshot shows a GitHub repository page for 'alfeoAntonioLuca / theSimpleClassifier'. The repository has 1 branch and 0 tags. A single commit from 'AntonLuca' titled 'Initial commit' was made 30 seconds ago. The commit message is 'Initial commit'. The files added in this commit are: .idea, README.md, irisclassifier.py, main.py, report.png, report.txt, requirements.txt, and test\_irisclassifier.py. The repository has 1 unwatched star, 0 forks, and 0 issues. The 'About' section notes 'No description, website, or topics provided.' The 'Readme' section links to the README.md file. The 'Releases' section indicates 'No releases published' and 'Create a new release'. The 'Packages' section links to 'Publish your first package'. The 'Languages' section shows Python at 100.0%.

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

master 1 branch 0 tags Go to file Add file Code

AntonLuca Initial commit 24e304e 30 seconds ago 1 commits

File	Message	Time
.idea	Initial commit	30 seconds ago
README.md	Initial commit	30 seconds ago
irisclassifier.py	Initial commit	30 seconds ago
main.py	Initial commit	30 seconds ago
report.png	Initial commit	30 seconds ago
report.txt	Initial commit	30 seconds ago
requirements.txt	Initial commit	30 seconds ago
test_irisclassifier.py	Initial commit	30 seconds ago

README.md

Unwatch 1 Star 0 Fork 0

About

No description, website, or topics provided.

Readme

Releases

No releases published  
Create a new release

Packages

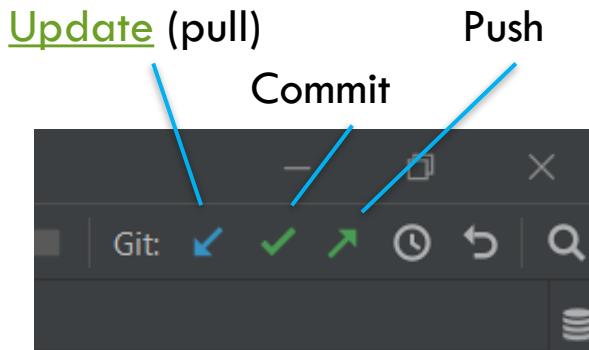
No packages published  
Publish your first package

Languages

Python 100.0%

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

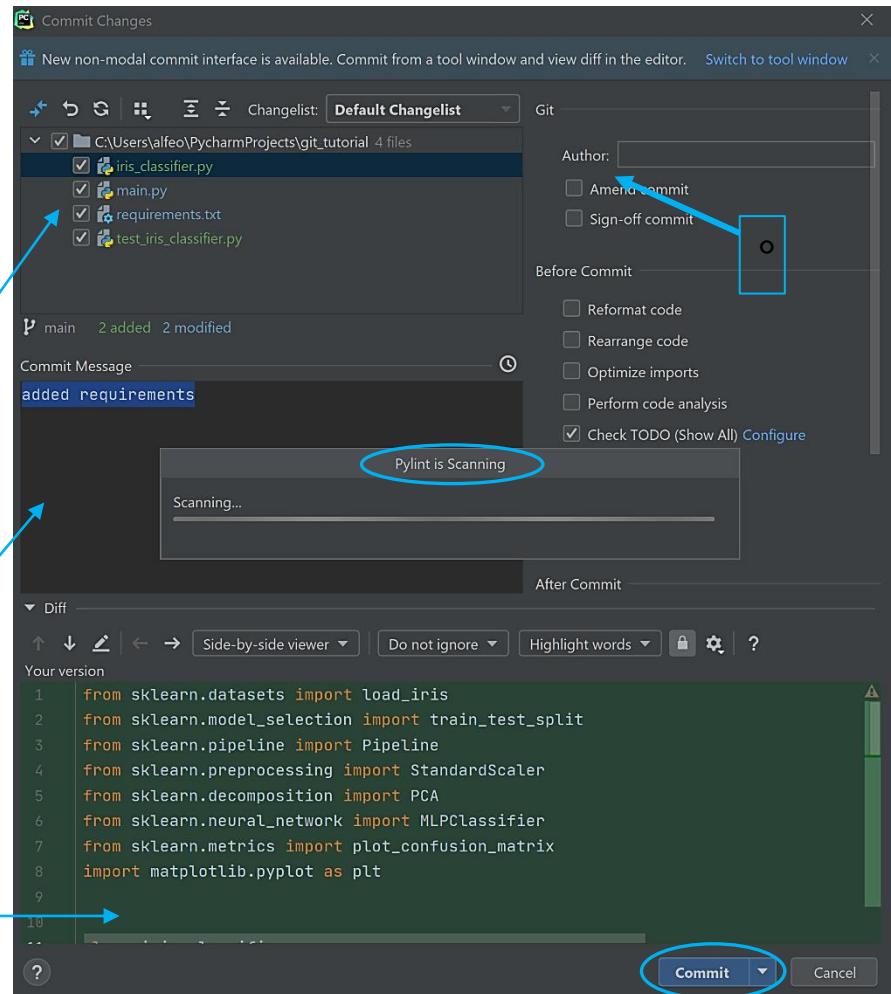


**Color code:**

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

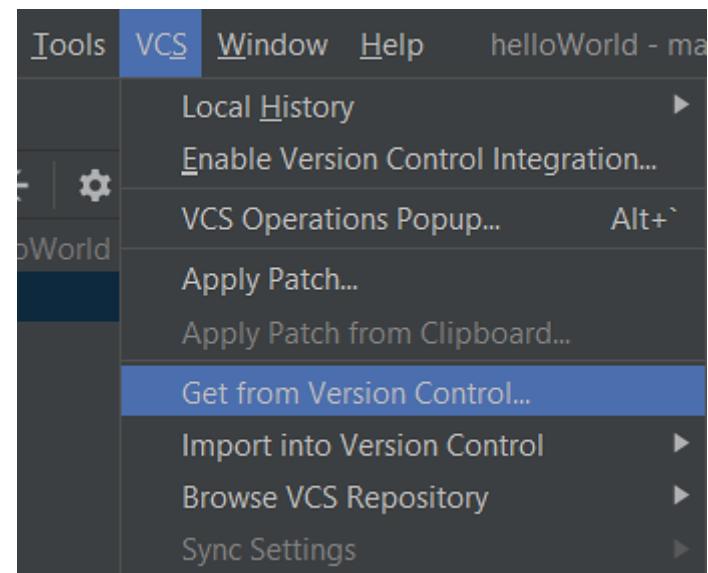
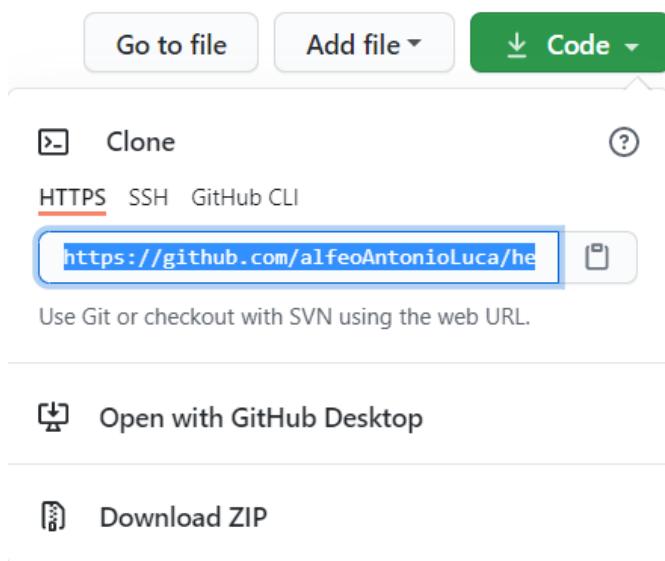
Explain clearly the change provided

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



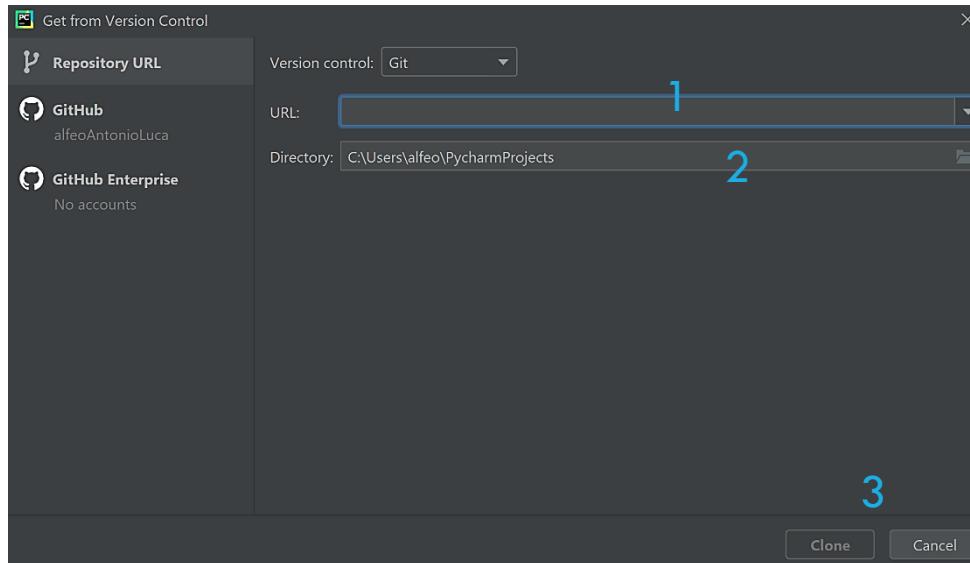
# CLONE FROM GIT 1/2

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 GitHub.

It allows you to contribute to a GitHub 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]