ML in Applications

Dipartimento di Automatica e Informatica
Politecnico di Torino, Torino, ITALY

Lab overview

Date	Time	Topic
17/4/2024	13:00-14:30	Milestone packages
17/4/2024	14:30-16:00	Time series classification
18/4/2024	10:00-11.30	Time series classification
24/4/2024	13:00-14:30	Image classification with textural features
24/4/2024	14:30-16:00	Image classification with bag of features
2/5/2024	10:00-11.30	Introduction to TensorFlow
8/5/2024	13:00-14:30	Transfer learning
8/5/2024	14:30-16:00	Transfer learning / HPC basics
9/5/2024	10:00-11.30	Uncertainty in deep learning
15/5/2024	13:00-14:30	Uncertainty in deep learning
15/5/2024	14:30-16:00	Time series anomaly detection
16/5/2024	10:00-11.30	Time series anomaly detection

Milestone packages

Numpy, Pandas, Matplotlib

Data Types in Python

- The **backbone of a Python object is a C structure**, which contains not only its value, but other information as well
- x = 10000 for instance defines a pointer to a C structure, which contains several values. Looking through the Python 3.4 source code, we find that the integer (long) type definition effectively looks like this:

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

- This means that there is **some overhead** in storing an integer in Python as compared to an integer in a compiled language like C

Data Types in Python

- A **Python list** Is more than just a list
- Thanks to Python's dynamic typing, we can even create heterogeneous lists:

```
In [5]:
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]

[bool, str, float, int]
```

- This flexibility costs! Each item in the list must contain its own type info, reference count, and other information
- The Python list contains a **pointer to a block of pointers**, each of which points to a full Python object

Numpy

Numpy - Intro

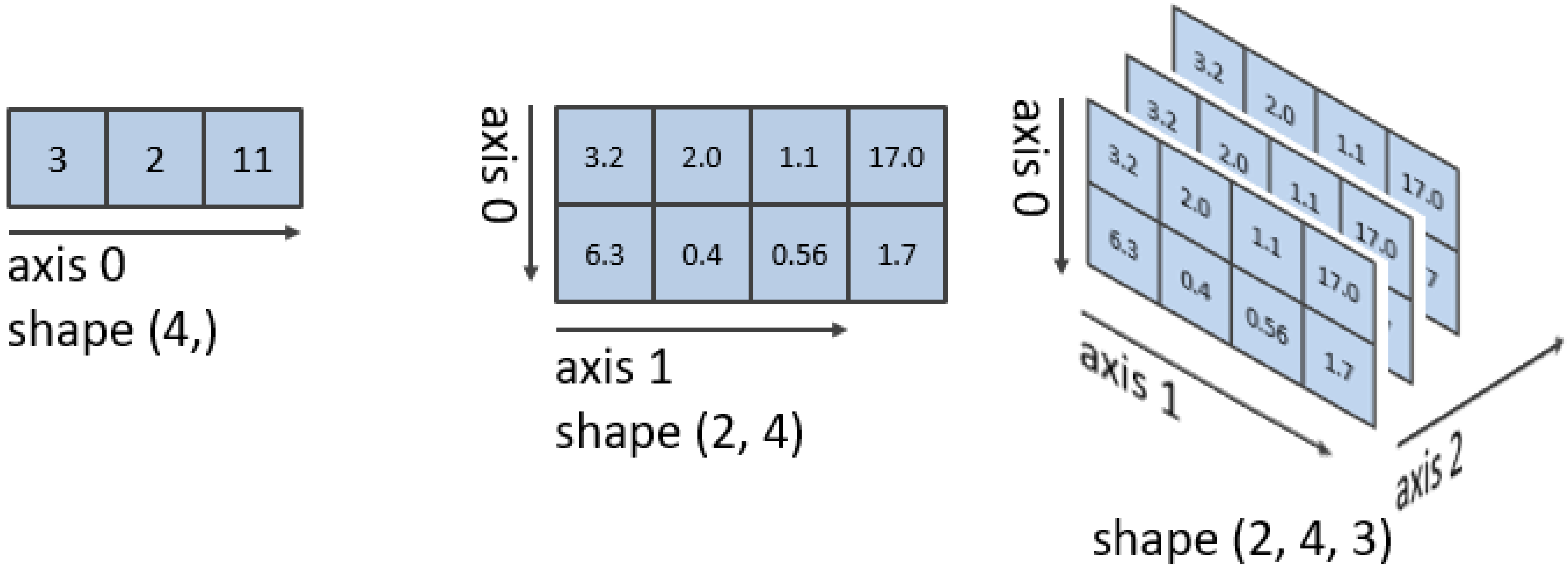
- Numpy provides the high-performance n-dimensional array object (*ndarray*) where:
  - all the elements must be of the same type (*dtype*)
  - the default *dtype* is float
  - the number of dimensions is the rank of the array
  - the shape of an array is a tuple of integers giving the size of the array along each dimension

- The *ndarray* exposes methods and attributes

- Python Lists vs. Numpy Arrays
  - Size: less space
  - Performance: faster than lists
  - Functionality: assortment of routines for fast operations on arrays

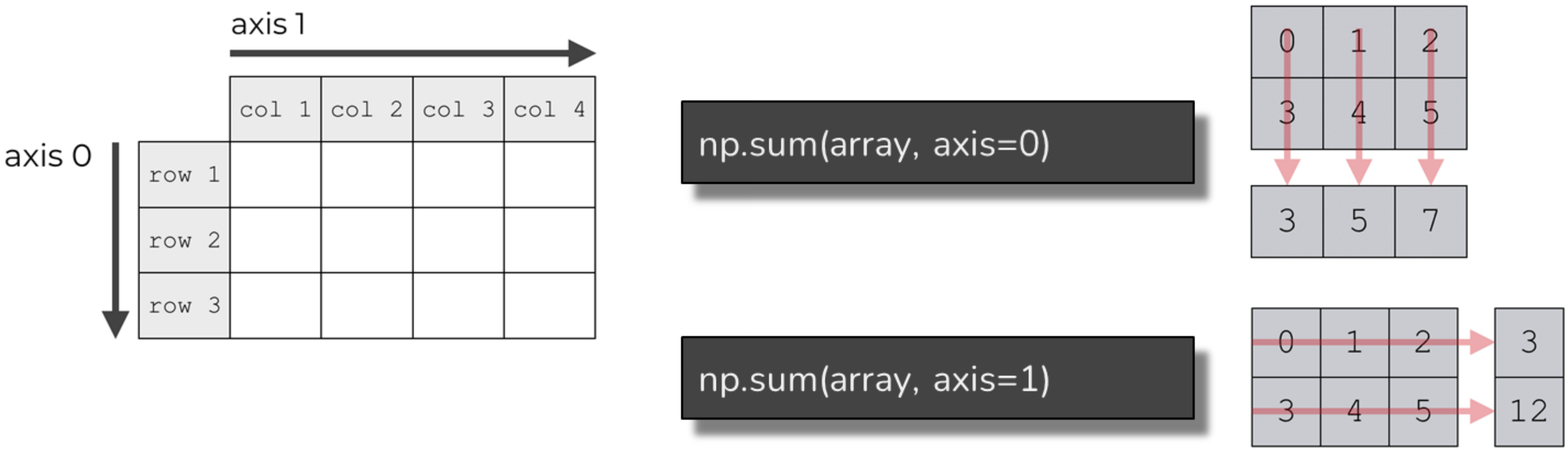
Numpy - Intro

- Data in *ndarrays* is in the **row-major (C) order**, unless otherwise specified
- The axis number of the dimension is the index of that dimension within the array's shape



Numpy - Axis

- We may perform **an operation along axis n of array** leveraging the **axis argument** of the array method



Numpy - Slicing

- Slicing: accessing sub-parts of an array
  - The **basic slice syntax is i:j:k** where i is the starting index, j is the stopping index, and k is the step (k!=0)

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]
array([1, 3, 5])
```

  - **Negative i and j** are interpreted **as n + i and n + j** where n is the number of elements in the corresponding dimension

```
>>> x[-2:10]
array([8, 9])
>>> x[-3:3:-1]
array([7, 6, 5, 4])
```

Numpy - reversing via slicing

Row-reversed

```
arr = [0 1 2 3 4 5 6 7 8]
Row-reversed:
arr[::-1] --> [8 7 6 5 4 3 2 1 0]
arr[len(arr)-1::-1] --> [8 7 6 5 4 3 2 1 0]
arr[-1::-1] --> [8 7 6 5 4 3 2 1 0]
arr[-1:0:-1] --> [8 7 6 5 4 3 2 1]
```

Column-reversed

```
arr =
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
arr[:, ::-1] =
 [[2 1 0]
 [5 4 3]
 [8 7 6]]
```

nD-reversed

```
[[[0. 0.]
  [0. 0.]]

 [[1. 1.]
  [1. 1.]]

 [[2. 2.]
  [2. 2.]]]

l.shape : (3, 2, 2)

l[::-1, :, :] =
[[[2. 2.]
  [2. 2.]]

 [[1. 1.]
  [1. 1.]]

 [[0. 0.]
  [0. 0.]]]
```

Numpy - indexing

- Integer array indexing (fancy indexing)
  - the row index is just [0, 1, 2] and the column index specifies the element to choose for the corresponding row, here [0, 1, 0]

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])
```

- Boolean array indexing
  - a common use case for this is filtering for desired element values: add a constant to all negative elements

```
>>> x = np.array([1., -1., -2., 3])
>>> x[x < 0] += 20
>>> x
array([  1.,  19.,  18.,   3.])
```

Numpy - View vs copy

- Unlike python lists, y = x[:] does not return a copy, it returns a **view**
- Array slicing returns a view
- Array indexing **copies** the data

```
In [21]:
import numpy as np
x = np.arange(10, dtype=np.int32)

print('An integer array:', x)
print('But if we view it as a float:', x.view(np.float32))
print("...It's probably not what we expected...")
```

```
An integer array: [0 1 2 3 4 5 6 7 8 9]
But if we view it as a float: [0.0e+00 1.4e-45 2.8e-45 4.2e-45 5.6e-45 7.0e-45 8.4e-45 9.8e-45 1.1e-44
 1.3e-44]
...It's probably not what we expected...
```

We are interpreting the underlying bits of the original memory buffer as floats

- **Avoid copies when using very large arrays**
- Use +=, -=, *=, etc to avoid making a copy of the array.
- x += 10 will modify the array in place (i.e., modifies the data-structure itself)
- x = x + 10 will make a copy and modify it (reassigns the variable)

Numpy - broadcasting

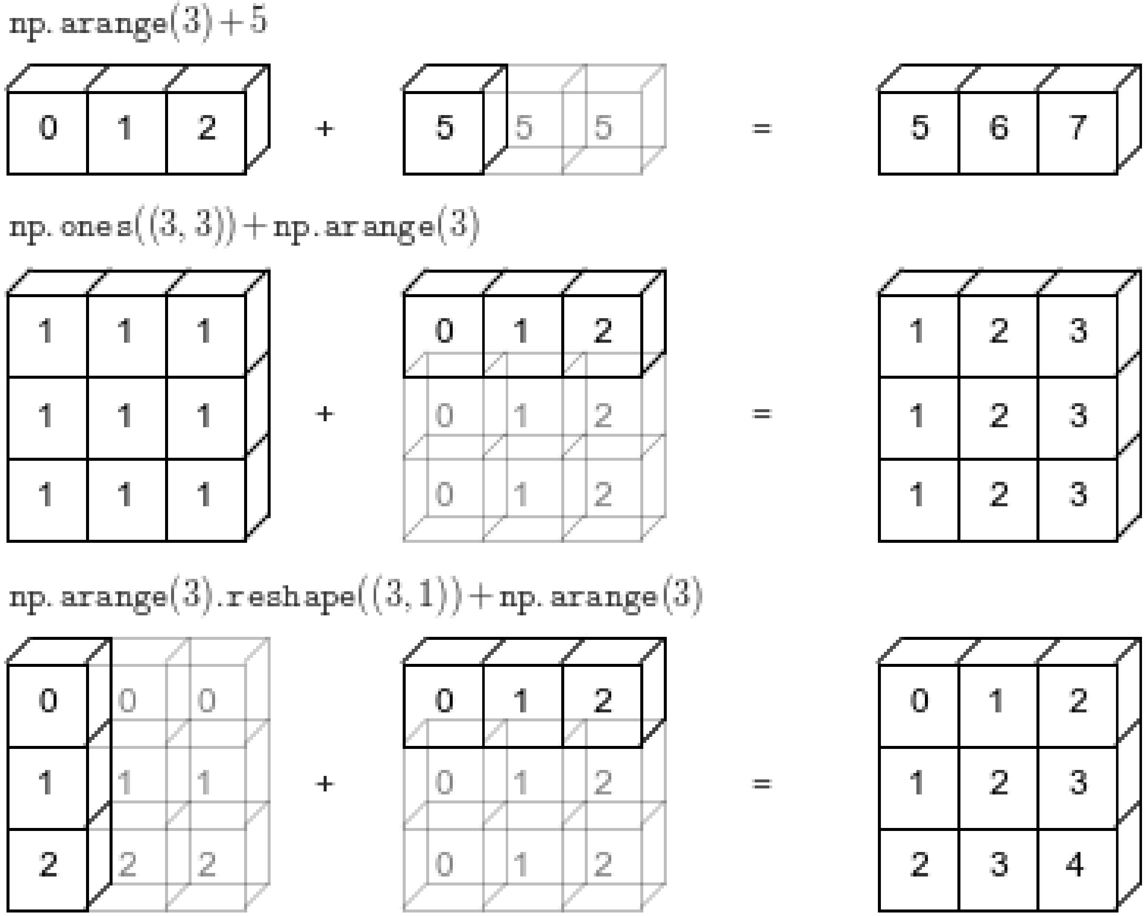
- For arrays of the same size, binary operations are performed on an **element-by-element basis**
- **Broadcasting allows** these types of binary operations to be performed **on arrays of different sizes**: a + 5
- We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5]
- This duplication of values does not actually take place, but it is a useful trick to think about broadcasting
- Case of adding a one-dimensional array to a two-dimensional array:

```
In [3]:
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b

array([5, 6, 7])
```

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)

[0 1 2]
[[0]
 [1]
 [2]]
```

```
a + b

array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Numpy - broadcasting

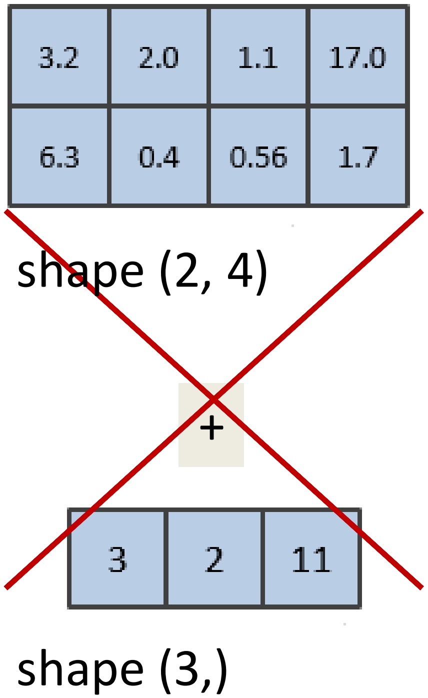


source: VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.".

Numpy - broadcasting

- Broadcasting in NumPy follows a strict set of rules. When operating on two arrays:
    - NumPy compares their shapes element-wise
    - It starts with the trailing (i.e., rightmost) dimensions and works its way left
    - Two dimensions are compatible when
        - they are equal, or
        - one of them is 1

If these conditions are not met, a *ValueError: operands could not be broadcast together* exception is thrown



- Why broadcasting? It provides a means of **vectorizing array operations** so that looping occurs in C instead of Python

Numpy - Vectorization

- For many types of operations, NumPy provides a **convenient interface into compiled routine**. This is known as a ***vectorized* operation**
- It consists in simply performing an operation on the array, which will then be applied to each element
- It is designed **to push the loop into the compiled layer that underlies NumPy**, leading to much faster execution
- Vectorized operations in NumPy are implemented via ufuncs, able to quickly execute repeated operations on values in NumPy arrays

Pandas

Pandas - Basics

- The backbone building block of pandas is the *Series*
  - Ordered collection of values, *generally* all the same type
  - Perfect way to collect lots of different observations of a variable
- The essential difference from *numpy* array is the presence of the **index**
  - The ndarray has an implicitly defined integer index
  - The Pandas Series has an explicitly defined index associated with the values (note that the index need not be an integer)
- The item access works as expected

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
```

```
data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
data['b']
0.5
```

Pandas - Basics

- The ***DataFrame*** is an analogy of a two-dimensional array with both flexible row indices and flexible column names
- You can think of a *DataFrame* **as a sequence of aligned Series objects**. Here, by "aligned" we mean that **they share the same index**
- Both Series and *DataFrame* can be **thought also as specialized dictionary**

```python
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

```python
states['area']
```

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Pandas - Indexing

- Pandas provides **array-style item selection via the same basic mechanisms as NumPy arrays**: slices, masking, and fancy indexing
- Pandas provides some **special indexer attributes** that explicitly expose certain indexing schemes:
  - The **loc** attribute allows indexing and slicing that always **references the explicit index**. If we reason in term of dict-style, we are accessing the key of the dict
  - The **iloc** attribute allows indexing and slicing that always references the **implicit Python-style index**

```
data = pd.Series(['a', 'b', 'c'],
                 index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
data.loc[1]
```

```
'a'
```

```
data.iloc[1]
```

```
'b'
```

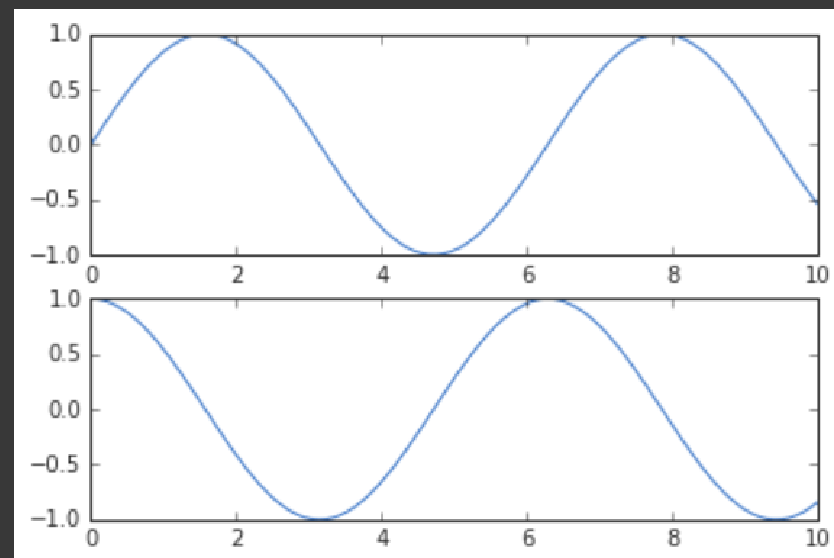
Visualization

Matplotlib - Intro

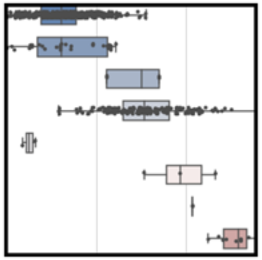
- Matplotlib plots the data on **Figures**, each one containing one or more **Axes**, an area where points can be specified in terms of coordinates

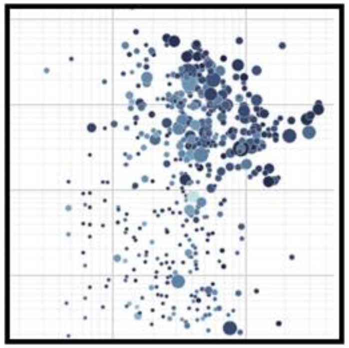
- A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a **more powerful object-oriented interface**

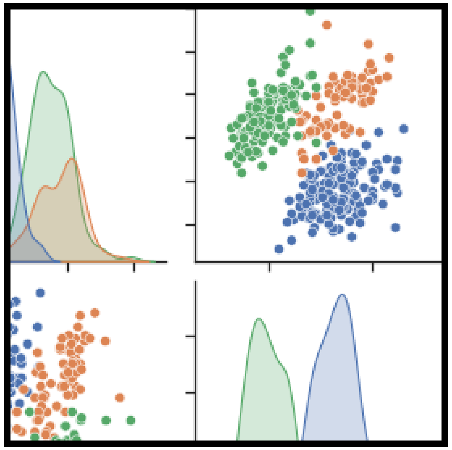
```python
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



Matplotlib vs. Seaborn vs. Plotly







Matplotlib:

- For the most basic exploratory analysis, directly integrated with Pandas

Seaborn:

- Essentially a prettier version of Matplotlib: good if you want exploratory analysis to look pretty for other people

Plotly:

- Focuses on **interactive visualization** and design for business intelligence

Tools – Jupter Notebook

- Why Jupyter?
  - ✓ One interface that can be used to run your analyses in several different programs: Jupyter: Ju (Julia) - py (Python) - teR (R)
  - ✓ Easy integration of text, code, and code output into a single document
  - ✓ Want to see the result of each line of code
  - ✓ Basics of Jupyter Notebooks here

- Why not Jupyter?
  - × It is almost impossible to enable good code versioning (notebooks are stored as big JSON files)
  - × There is no (good) IDE integration
  - × Very hard to test (especially with TensorFlow)

Excellent tools for exploration, not for production

Python and Jupyter

- **Ubuntu 22.04.4 LTS**
  - Virtual machine (VMware Workstation Player, Virtual box)
  - Windows Subsystem for Linux (WSL)
- **Virtual environment setup**
  1. Create a virtual environment
  2. Activate a virtual environment
  3. Upgrade pip
  4. Install packages via requirements.txt
  5. Leave the virtual environment

```
python3 -m venv my_env
source my_env/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
deactivate
```

Python and Jupyter

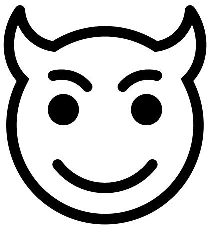
- **Ubuntu 22.04.4 LTS**
  - Virtual machine (VMware Workstation Player, Virtual box)
  - Windows Subsystem for Linux (WSL)
- **Virtual environment setup**
  1. Create a virtual environment
  2. Activate a virtual environment
  3. Upgrade pip
  4. Install packages via requirements.txt
  5. Leave the virtual environment

requirements.txt

```
numpy
pandas
scikit-learn
matplotlib
seaborn
jupyter
jupyter_contrib_nbextensions
jupyterthemes
requests
```

Python and Colab

- Colab: de-facto a Jupyter notebook online runnable, leveraging Google’s hardware
- Google will probably steal our soul …
- *But* too convenient and easy to use
- *But* we have a lot of online resources
- *But* good integration with drive
- *But* we need GPUs!
  - **GPU computation available**, even if restricted



- So … please, here my soul

Labs rules

- For each Lab session, you will receive an assignment in form of Jupyer Notebook file (.ipynb)
- To be evaluated, such file must be uploaded on the Portale della didattica → MLinAPP → Elaborati **by the end on the lab**
- No delay will be admitted
- You will receive the results at the end of the course, before the exam session