Introduction to python for AI

Nicolás Pascual

April 2019

Academy AI

About myself

- **Name** Nicolás Pascual
- **Background** Software Engineer specially interested in good practices and code quality.
- **Studying** Currently studying a masters degree on artificial intelligence.
- **Contact** nicolaspascualgonzalez@gmail.com
- **Github** @nicolaspascual
- **Linkedin** Nicolas Pascual

Installation

Installation

There are several ways to install python depending on the OS. For matter of simplicity we will stick to Conda which will handle for us virtual environments and the installation of all the desirable dependencies for machine learning.

Depending on the OS and the distribution the installation is different, in the web page of anaconda you will find all the instructions. Anaconda

Introduction to IPython and Jupyter

IPython

Ipython is an interactive console that adds some functionalities that allows us to develop with ease.

To run it we just need to run `ipython` in the console. A shell will prompt and we will be able to execute commands from there on. Some of the features of IPython are:

1. Tab completion.
2. Run multiline commands.
3. Introspection (*dir*).
4. Run commands from the clipboard with `\%paste`.
5. Run commands from another file `\%run`.
6. Beautiful printing stack traces and outputs.

Magic commands I

The shell brings some utility commands that will allow us to perform some actions such as timing or profiling easily. A list of those can be seen here.

- **%quickref** Display the IPython Quick Reference Card
- **%magic** Display detailed documentation for all of the available magic commands
- **%debug** Enter the interactive debugger at the bottom of the last exception traceback
- **%hist** Print command input (and optionally output) history
- **%pdb** Automatically enter debugger after any exception
- **%paste** Execute pre-formatted Python code from clipboard
- **%cpaste** Open a special prompt for manually pasting Python code to be executed
- **%reset** Delete all variables / names defined in interactive namespace

Magic commands II

- **%page OBJECT** Pretty print the object and display it through a pager
- **%run script.py** Run a Python script inside IPython
- **%prun statement** Execute statement with cProfile and report the profiler output
- **%time statement** Report the execution time of single statement
- **%timeit statement** Run a statement multiple times to compute an emsemble average execution time. Useful for timing code with very short execution time
- **%who, %who_ls, %whos** Display variables defined in interactive namespace, with varying levels of verbosity
- **%xdel variable** Delete a variable and attempt to clear any references to the object in the IPython internals

Running shell commands

This environment let us run shell commands from the OS by using ! before a statement, so we could for example see the current directory or change it.

```
files =!ls
files # main.py, .git
```

Jupyter notebooks

On top of IPython emerges Jupyter allowing us to run IPython in the browser with rich outputs and a lot of benefits. To span a jupyter process you just have to do the following:

```
$ jupyter notebook
```

Introduction to NumPy

NumPy

Numpy is the standard in python for high performance scientific computations. Its main features are:

1. **ndarray**
2. **Mathematial operations for fast computation**
3. **IO operations**
4. **Linear Algebra**
5. **C/C++ Integration and communication**

Importing and installing numpy

Numpy comes installed with the conda enviroment. In other scenario we need to install it by means of pip, to install it we just need to run:

```
pip install numpy
```

Numpy is a 3rd party module of python and is imported as so. By convention we usually import numpy with the special name np for matter of simplicity, the final import statement usually looks as follows.

```
import numpy as np
```

ndarray

Ndarrays (n-dimensional arrays) are a fast, flexible container for large data sets in python. This kind of arrays allows you to perform mathematical computations over whole blocks of data using similar syntax to the one used on single elements

```
data =np.array([1, 2, 3])

data +data # 2, 4, 6
data *data # 1, 4, 9
```

Ndarray creation I

The easiest way to create a numpy array is to use a standard python array and pass it to the constructor, multidimensional arrays are created as you may suspect by passing nested arrays to the constructor.

```
np.array([1, 2, 3]) # array([1, 2, 3])
np.array([[1, 2, 3], [1, 2, 3]]) # array([[1, 2, 3],[1, 2, 3]])
```

To see what is the size and the dimension of a ndarray we can use the methods `ndim` and `shape`.

```
arr =np.array([[1, 2, 3], [1, 2, 3]])
arr.ndim # 2
arr.shape # 2, 3
```

Ndarray creation II

Numpy provide us some special creation methods for convenience such as `ones` and `zeros`. These methods come really handy when you have to create a placeholder for later adding values.

```
np.zeros(3) # array([0., 0., 0.])
np.ones((2, 3)) # array([[1., 1., 1.], [1., 1., 1.]])

np.arange(10) # array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Ndarrays data types I

For performance matters numpy lets us define the data type (dtype) of our array by passing the named argument `dtype`, with this argument we are able to tell numpy how big is the chunk of memory that it needs.

If this information is not present on creation numpy will infer it from the actual data. Data type may be changed once the array has been created with `astype`.

Some of the most used dtypes are **int8...64**, **uint8...64**, **float8...64**, **bool** or **string**.

Ndarrays data types II

```
arr =np.array([1, 2, 3])
arr.dtype # int64

float_arr =arr.astype(np.float64)
float_arr.dtype # float64

bool_arr =np.array([1, 1, 0], dtype=np.bool)
bool_arr # array([True, True, False])
```

Indexing and Slicing I

To access a specific item or a set of them you can simply use `[ ]` as you would do on a normal array keep in mind that slicing will create a view of the original array and therefore any mutation performed on the slice will end up in the original array.

```
arr =np.array([1, 2, 3, 4])
arr[0] # 1
arr[0:2] # array([1, 2])
arr[0:2] =0
arr # array([0, 0, 3, 4])
```

To avoid working on a view you can perform a `copy` of the original array.

```
arr =np.array([1, 2, 3, 4])
new_arr =arr[0:2].copy()
new_arr =0
arr # array([1, 2, 3, 4])
```

Indexing and Slicing II

To index over multiple dimension you can just pass comma separated integers to the selector.

```
arr =np.array([[1, 2], [3, 4]])
arr[0, 0] # 1
```

Multiple slices can also be passed to select a whole column for example.

Multiple elements can be picked independently by passing an array of integers to the selector (**fancy indexing**)

```
arr =np.array([[1, 2], [3, 4], [5, 6]])
arr[:, 0] # array([1, 3, 5])
```

Boolean indexing

Boolean indexing consist in passing to the selector a boolean array with the same dimensions of the original array, the places where there is a `True` will be selected.

This feature together with the facilities of numpy to execute logical operations over the array makes filtering a lot more easy.

```
arr =np.array([1, 2, 3, 4, 5, 6])
arr % 2 ==0 # array([False, True, False, True, False, True])
arr[arr % 2 ==0] # array([2, 4, 6])
```

Transposing arrays

A very common operation when performing algebra is to do transposing, in numpy is very easy to perform this operation by using the `T` property of the ndarrays.

```
arr =np.arange(4).reshape((2, 2)) # [[0,1],[2,3]]
arr.T # [[0,2],[1,3]]
```

Exercise

Given the matrix T:

$$\begin{vmatrix} 3 & 10 & 4 \\ 21 & 11 & 33 \\ 12 & 1 & 0 \end{vmatrix}$$

Calculate $T * T'$

Element-wise operations

As seen before numpy bring a lot out of the box algebraic functionality. Some examples are: `abs` `sqrt` `isnan` `mod` `mean` `min`, `max` `sum` `any` `all`

```
arr =np.array([np.nan, 0, 1])
np.isnan(arr) # [True False False]

np.isnan(arr).all() # False
np.isnan(arr).any() # True
```

As 80% of data science is preprocessing these kind of function will come very handy when we face this part of the process.

Exercise

Given the previously defined product $T * T'$. Check if there is any even number and calculate the sum among them.

Meshgrids

When having to create an artificial dataset we usually rely on functions and sampling. Numpy makes this task really easy by means of `meshgrid` and element-wise operations. Meshgrid returns coordinate matrices from coordinate vectors. This example is to replicate the function $z = \sin(x) + \cos(y)$.

```
points =np.arange(-10, 10, 0.1) # equally spaced points
xs, ys =np.meshgrid(points, points)
zs =np.sin(xs) +np.cos(ys)
```

Random number generation I

Numpyu provides an easy standad way of generating random samples with a specific size and a specific distribution (`normal`, `binomial`, `beta`, `chisquare`, `gamma` or `uniform`).

```
np.random.normal(size=(4, 4))
np.random.uniform(size=10)
```

Inside the random package we also can find methods for shuffling (`np.random.shuffle`) and making permutations (`np.random.permutation`).

Array reshaping

Some times we need to change the dimensions of an array to match the specific input of a method (neural networks). Numpy provides the method `reshape` for this purpose.

```
arr =np.arange(8) # array([0, 1, 2, 3, 4, 5, 6, 7])
arr.reshape((4, 2)) # array([[0, 1],
                    # [2, 3],
                    # [4, 5],
                    # [6, 7]])
```

The inverse operation is called flatten and is performed in the same fashion `arr.flatten()`

Sorting

Sorting in numpy is performed by the method `sort`. Depending on how you call this method the sorting will be in place or returning a copy, while `np.sort` returns a copy `arr.sort()` does the operation inplace.

```
arr =np.random.normal(8) # array([0, 1, 2, 3, 4, 5, 6, 7])
arr.sort() # inplace
```

Exercise

Compute a random vector of length 100 and get the value at the 25 quantile

Extra

Notebook

Introduction to Pandas

Pandas

Pandas is, probably, the most useful tool that you have in python for managing data. It helps you working with data keeping the benefits of numpy behind the hood. Pandas provide a number of features.

- Data structures **with labeled axes**.
- Date time handling.
- Element-wise operations.
- Comfortable handle of missing values.
- SQL-like operations. (Merging, ...)

Importing and installing pandas

Pandas comes installed with the conda enviroment. In other scenario we need to install it by means of pip, to install it we just need to run:

```
pip install pandas
```

Pandas is a 3rd party module of python and is imported as so. In this case we usually import pandas with the alias pd.

```
import pandas as pd
```

Series I

A series is a one dimensional data structure, a **vector**. It contains an array of values, an ndarray, and a set of labels (index) associated to those values.

```
pd.Series([4, 6, -5, 3])
# 0 4
# 1 6
# 2 -5
# 3 3
# dtype: int64

pd.Series([4, 6, -5, 3], index=['x1', 'x2', 'y1', 'y2'])
# x1 4
# x2 6
# y1 -5
# y2 3
# dtype: int64
```

Series II

To access a value in the Series we will simply rely on the common `[ ]` and the names of the indexes we want to access.

```
bounding_box =pd.Series([4, 6, -5, 3], index=['x1', 'x2', 'y1'
                                              , 'y2'])
bounding_box['x1'] # 4
bounding_box[['x1', 'y1']]
# x1 4
# y1 -5
# dtype: int64
```

Indexing **are** views of the original Series. Indexes are conserved and modifications are performed on the original object.

Series III

Most of the operations that we explored with numpy are accessible from the pandas interface. This is very handy to perform algebraic and element-wise operations.

```
bounding_box =pd.Series([4, 6, -5, 3], index=['x1', 'x2', 'y1'
                                              , 'y2'])

bounding_box /bounding_box.sum()
# x1 0.500
# x2 0.750
# y1 -0.625
# y2 0.375
# dtype: float64
```

Operations will preserve the index.

Series IV

Very typically we will be hunting missing values inside our data. Pandas back us up with the method `pd.isnull()`

```
missing_data =pd.Series([4, 6, -5, None], index=['x1', 'x2', '
                                        y1', 'y2'])

pd.isnull(missing_data)
# x1 False
# x2 False
# y1 False
# y2 True
# dtype: bool
```

Operations will preserve the index.

Exercise

Normalize to a 0 - 1 range the Series obtained by running

```
pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/
                                                adult/adult.data',
    header=None, usecols=[0]
)
```

Note:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

DataFrame

DataFrames are a collection of series, forming a bidimensional matrix. Indices of the series represent the columns of the dataframe.

```
data ={
    'name': ['Maria', 'Carla', 'Juan', 'Ana', 'Sergio'],
    'age': [15, 33, 12, 21, 45],
    'gender': [True, True, False, True, False]
}

pd.DataFrame(data)
# name age gender
# 0 Maria 15 True
# 1 Carla 33 True
# 2 Juan 12 False
# 3 Ana 21 True
# 4 Sergio 45 False
```

The most common way to create a dataframe for is to use a dictionary of equally sized arrays in which every key represent a column.

IO Operations

In the vast majority of the cases we will be loading a csv from our filesystem, pandas works transparently for us in this sense by means of `read_csv`.

```
pd.read_csv('path/to/csv.csv')
```

This method is very customizable, refer to the documentation for more details. Documentation.

To ouput to a file we can use analogously the method `to_csv` which will require the path to save the file.

```
df.to_csv('path/to/csv.csv')
```

Index Object

Every row in a Dataframe has its own index assigned, the Index of a DataFrame is formed by a list of object assigned to each row.

This comes specially useful when we have to filter the data or divide them as we will maintain a way to refer to each specific row.

To access the index of a DataFrame we can call the method `index`, this object is inmutable.

```
df =pd.DataFrame(data)
df.index # Int64Index([1, 2, 3, 4, 5], dtype='int64')
```

ReIndex

You have the possibility of reindexing a dataframe by accessing the property `index` and passing an index.

```
df =pd.DataFrame(data)
df.index =['m','c','j','a','s']
df
# name age gender
# m Maria 15 True
# c Carla 33 True
# j Juan 12 False
# a Ana 21 True
# s Sergio 45 False
```

Indexing and Slicing I

There are two ways to index a DataFrame.

1. **iloc** Accesses the dataframe by row position not taking into account the index.
2. **loc** Accesses the dataframe by index.

```
a =pd.DataFrame(data, index=['m','c','j','a','s'])

a.loc['m'] ==a.iloc[0] # True
```

Indexing and Slicing I

Slicing is performed in a similar fashion. Note that you can slice using labels and it will take all elements in between both labels.

```
a =pd.DataFrame(data, index=['m','c','j','a','s'])

a.loc['m':'c']
# name age gender
# m Maria 15 True
# c Carla 33 True
```

Dropping

Pandas lets us drop columns or rows. To do so we just need to call `drop` with the labels or columns and the axis, 0 for rows and 1 for columns. The method performs a copy instead of doing it inplace

```
df =pd.DataFrame(data, index=['m','c','j','a','s'])

df.drop('name', axis=1)
# age gender
# m 15 True
# c 33 True
# j 12 False
# a 21 True
# s 45 False
a.drop(['m', 'c'])
# name age gender
# j Juan 12 False
# a Ana 21 True
# s Sergio 45 False
```

Dropping duplicates

In same scenarios we will face duplicated samples to get rid of them we can rely on `drop_duplicates`. This method will remove at a row level the duplicates.

```
df =pd.DataFrame({'f1': [1, 2, 2, 1], 'f2': [0, 1, 1, 1]})
df.drop_duplicates()
# f1 f2
# 0 1 0
# 1 2 1
# 3 1 1
```

Exercise

Remove the first 50% of the df

```
pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/
                                                adult/adult.data',
    header=None, usecols=[0]
)
```

Filtering

Analogously to numpy pandas allows filtering by boolean indexing. In this sense we can perform check as the following ones.

```
df =pd.DataFrame(data, index=['m','c','j','a','s'])
df[df.gender ==False]
# name age gender
# j Juan 12 False
# s Sergio 45 False
```

Arithmetic Operations

In the same manner we did with numpy we can perform arithmetic operations over dataframes preerving the indexes.

```
df +df
df /df
df *df
```

We can also do these operations among dataframes and series.

Custom function application

Element wise operations can be performed throughout the dataframe easily be means of the `apply` method.

```
df['gender'] =df['gender'].apply(lambda x: not x)
```

The operation is not performed in place and for it to take place we need to assign it to the original dataframe.

Sorting

We can sort a Dataframe with the method `sort_index`, by providing the column we will be able to create a sorted copy of the dataframe.

```
df.sort_index(by='age')
# name age gender
# j Juan 12 False
# m Maria 15 True
# a Ana 21 True
# c Carla 33 True
# s Sergio 45 False
```

Statistics

Computing statistics from a dataframe is very easy thanks to the ability of pandas to handle functions. It also provides some utility methods for convenience (`count`, `min`, `max`, `std`, `mean`, `median`). A statistical summary of the dataframe can be obtained by means of `describe`.

```
df.describe()
# age
# count 5.000000
# mean 25.200000
# std 13.682105
# min 12.000000
# 25% 15.000000
# 50% 21.000000
# 75% 33.000000
# max 45.000000
```

Value counts

In a pandas series we can count the number of appareances of the values by means of `value_counts`.

```
df['gender'].value_counts()
# True 3
# False 2
# Name: gender, dtype: int64
```

Filling missing values

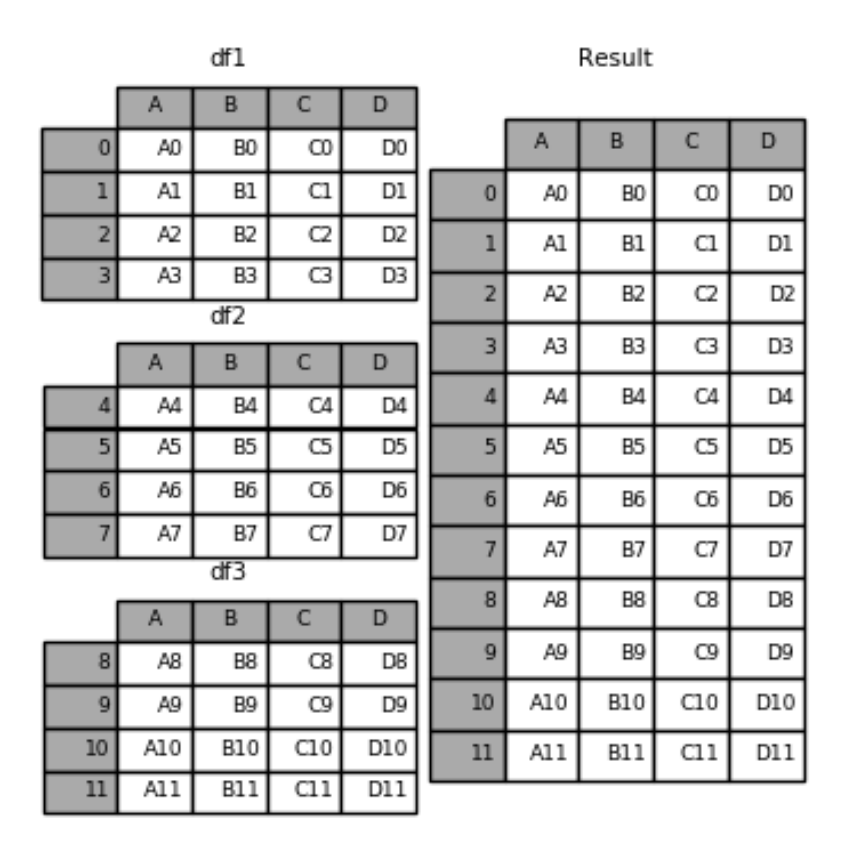
Sometimes we encounter missing values in our data, we have seen how to detect them, but to fill them with a imputation technique we can use the method `fillna`

Exercise

Choose a dataset from your own choice, you can take a look at the ones that UCI provides and look for the missing values, replace them by the mean and compute some statistics.

Joining vertically

When dealing with several files or dataframes we may have the data splitted in several ones. To join them we can use `pd.concat`.



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Joining horixontally

In other scenarios we may have to deal with having the dataset splitted by columns in this cases we can use aswell `concat` but along the axis 1.

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

This kind of operations have a lot of variants I recommend taking a look to the documentation for more information. Documentation

Extras

Notebook