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

Getting to know you

Google Form

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 Python

Python

Python is a dynamic interpreted language specially designed for convenience a fast development. For this reason it has been highly adopted by a lot of scientific communities such as data scientist.

Python is not the most efficient of the languages as it is interpreted but its high readability makes it very convenient for a most of applications.

In this course we will be using Python 3.6 (Python 2.7 is discontinued at last).

Indentation

The first thing that you notice when entering python is that it uses indentation as part of the definition of the language. Different scopes are denoted by their indentation. By convention use **4 spaces** not tabs.

```
for i in [1, 2, 3]:
    print(i) # Four spaces
```

As seen on top one line comments start with the special character #. Multi-line comments start and end with """

```
def func():
    """
    Example of pydoc
    """
    return None
```

Dynamic language I

Python is a dynamic language, this means that variables are not typed, this allows us to change the type in run time of the variables or to pass different types of parameters to the same function.

```
l =[1, 2, 3]
l.append(4) # int
l.append(5.0) # float
l # [1, 2, 3, 4, 5.0]
```

Python is taking care behind the scenes of changing the type of basic elements when needed.

```
1 /2 # 0.5
```

This have a very important implication which means that you do not have to care of managing memory not even for overflowing issues.

Dynamic language II

Variables can be forced to cast whenever is needed, casting is python is done as follows.

```
a =1
str(a) # "1"
```

We can check for the type of a variable. This allows us to check if a variable is from a class and therefore has certain functionality.

```
a =1
type(a) # int
a =str(a)
type(a) # str
isinstance(a, str) # True
isinstance(a, int) # False
```

Built-in types

Basic built in types are the following ones. More complex data types as list have their on literals but their functionality in the standard library.

Type	Description
None	'null' value
str	String
unicode	Unicode
float	Double-precision
bool	Logical
int	Signed
long	Arbitrary

Function Declaration I

Functions in python are declared with the keyword `def` and can be defined in the global scope (not inside classes)

```
a =1
def func(b, c =2):
    return a *b *c
```

As you can see functions can receive any number of parameters and can access their outer scope. Keep in mind that functions are *first class citizens*, this means that can be passed as parameters, stored and referenced straightforward.

Function Declaration II

In python parameters are passed through **reference**, this means that any mutation performed on the parameter changes the passed object.

```
def append_element(l, e):
    l.append(e)

arr =[1, 2, 3]
append_element(arr, 4)
arr # [1, 2, 3, 4]
```

Lambda Functions

In python we can create anonymous functions with a one-liner by means of the `lambda` expression.

```
f =lambda x, y: x +y
f(1, 2) # 3
```

Flow control: ifs

If statements allow to control the flow of our code. In python ifs are defined as follows:

```
a =int(input())
if a % 2 ==0:
    print('Multiple of 2')
elif a % 3 ==0:
    print('Multiple of 3')
else:
    print('Not multiple of 2 neither 3')
```

Flow control: loops

For loops iterate over the elements of the iterables.

```
for ch in ['a', 'b', 'c']:
    if ch =='b':
        continue
    print(ch)
# a c
```

While loops are defined in the following fashion:

```
a =10
while True:
    if a ==0:
        break
    print(a)
    a -=1
# 10 9 8 7 6 5 4 3 2 1
```

Exception handling

We can control the throwed exceptions by means of `try` and `except`. This allows us to catch the exceptions and continue the execution of the program.

```
try:
    callMethod()
except Exception:
    print('Error catched')
```

Class Definition

Python is an object oriented programming language and therefore has its own form of defining classes with the special keyword `class`.

```
class MyClass(object): # inherits from object (optional)
    multiply_by =2 # Class variable

    def __init__(self, a): # constructor
        self.a =a # instance variable

    def f(self): # instance method
        return self.a *MyClass.multiply_by
```

Function inside classes need to have as first parameter `self`, this means that the function is an instance function and allows the function to access the properties of the object.

Class instantiating

Instantiating a class is straightforward (no need of new):

```
obj =MyClass(2)
```

Function invocation is just as easy

```
obj.f() # 4
```

Exercise

Create a class that holds the information of person (name, surname, born_date and gender).

Numerics

Numeric types come with a bunch of methods worth mentioning:

- `a + b` Add a and b
- `a - b` Substracts a and b
- `a * b` Multiply a by b
- `a / b` Divide a by b
- `a // b` Floor-divide a by b, dropping any fractional remainder
- `a ** b` Raise a to the b power

Logival values

Python handles a binary logic around `True` and `False`. Common operations are performed with `or`, `and` and `not`.

```
True and False # False
True and True # True
True or False # True
True or not False # True
```

Python as many other languages can evaluate to a boolean all the objects, for example iterables will be evaluated as True when their size is greater than 0.

Tuples

Tuples are a one dimensional inmutable list capable of storing objects. Any kind of iterator can be transformed to a tuple.

Unpacking is a handy feature of tuples that allows us to extract the variables into separate lists.

```
a =1, 2
b, c =a # unpacking
b # 1
c # 2
```

Lists I

List in python are defined as literals and they are **mutable**. To access the array similarly to other languages we use bracket (`[ ]`) indexing.

A very handy tool offered by python is **slicing**, we can select a subset of the array by using the special notation `[start:stop:step]`.

```
l =[1, 2, 3]
l[0] # 1
l[0:2] # [1, 2]
l[2:0:-1] # [3, 2]
l[-1] # 3
```

Lists II

Common operations on lists are:

- `l.append(e)`: Adding element.
- `l.pop()`: Remove last element.
- `len(l)`: Gets the size of the list.
- `l.sort()`: Sorts the list.
- `e in l`: Checks for an element in the list.
- `l1 + l2`: Adds both lists

For more information address to the official documentation.

Dictionaries I

Dictionaries are a special kind of collection that allows us to index by an object. Python allows us to declare them in a literal fashion.

To access the array we will also use brackets ([ ]).

```
dict ={
    'a': 1,
    'b': 2,
}

dict['c'] =3
dict['c'] # 3
dict['a'] # 1
```

Dictionaries II

Common operations on dictionaries are:

- `k in d`: Checks for a key in the dictionary.
- `len(d)`: Gets the size of the dictionary.
- `d.values()`: returns a list of the values.
- `d.keys()`: returns a list of the keys.
- `d.items()`: returns a list of tuples as `(key, value)`

Important to note that we can iterate over the dictionary in a fancy way by transforming it to tuples and unboxing them.

```
for key, value in d.items():
    print(key, value)
# a, 1 b, 2 c, 3
```

For more information address to the official documentation.

Sets

Sets are unordered lists without repetition, they come specially handy when we need to perform operations such as join or intersection.

Common operations on dictionaries are:

- `e in s`: Checks for an element in the set.
- `s1 & s2`: Performs the intersection of the sets
- `s1 | s2`: Performs the union of the sets
- `s1 - s2`: Performs the difference of the sets
- `len(s)`: Gets the size of the set.

```
a =set([1, 2, 3])
b ={2, 3, 4}

a.union(b) # or a & b => 1, 2, 3, 4
a.intersection(b) # or a | b => 2, 3
```

Strings I

Strings in python are very similar to a list of chars but they have their own functionality and utility methods.

You can index a string just like you would index a list `[ ]`. You can also check for the presence of another string by using the same interface as with lists `in`.

```
s ='Hello'

s[0] # 'H'
s[0:2] # He
'el' in s # True
```

Strings II

String interpolation is one of the things that come really handy one digging into code. Python 3.6 brings the possibility of using a special kind of strings, the **f-strings** that come with a very nice interface.

```
a =1
s =f'A is {a}'
s # A is 1

s ='A is ' +str(a) # we need to cast the integer
```

Strings III

Usually we will be dealing with strings, python provides with a set of methods to treat them, some of them are very useful.

```
s ='Hello world! '
s.split(' ') # ['Hello', 'world!', '']

s.strip().split(' ') # ['Hello', 'world!']
```

Exercise

Add to the class created earlier a method that represents the object as a string in the following way:

'Surname, Name - Age - Gender'

Comprehensions

One of the better features of python is comprehension, with this feature we can easily create and filter complex lists, sets or dictionaries.

We will use an example to illustrate it:

```
l =[1, 2, 3]
pows =[i**2 for i in l]
evens =[i for i in l if i % 2 ==0]

dict ={str(i): i for i in l}
```

Handling files

Python lets us read, write and append files easily with the `open(filename, mode)` command. Depending on the mode we tell the OS how we want to open the file.

- **r** Reading.
- **r+** Reading and Writing with position at beginning.
- **w** Writing erasing file.
- **w+** Writing and Reading erasing file.
- **a** Appending to the file.
- **a+** Appending and reading the file.

```
with open('filename', 'r+') as file_handler:
    file_handler.readlines()
```

Mutability

Most objects in python are **mutable** with some exceptions.

```
a =[1, 2, 3]
a[0] =2
a # [2, 2, 3]
```

Tuples for example are inmutable objects.

```
a =(1, 2, 3)
a[0] =2 # TypeError: 'tuple' object does not support item
                                        assignment
```

Special Methods I

Python provides us the opportunity to implement some special methods and add our classes to complex behaviours such as generators, string representations or comparisons. The following are some examples:

1. `__repr__`: Similar to to_string
2. `__gt__ __lt__` greater than and less than respectively
3. `__ge__ __le__` greater than or equal and less than or equal respectively
4. `__eq__ __ne__` equal and not equal respectively-
5. `__add__` add.

Special Methods II

With this behaviour we can do operator overflowing hiding logic transparently and creating a fluent API.

```
class Point:
    def __init__(self, x, y):
        self.x =x
        self.y =y
    def __add__(self, other):
        x =self.x +other.x
        y =self.y +other.y
        return Point(x,y)
p1 =Point(2,3)
p2 =Point(-1,2)
p1 +p2 # (1,5)
```

Module structure

Every python file creates a module, a module is a set of function or classes that can be imported from other files. This lets us divide our logic into semantically similar functionalities.

Typically we will implement a class or a set of methods per file and then import them from other files.

```
#----------------------file1
def f(a, b):
    return a *b
#----------------------file2
from file1 import f
f(2, 2) # 4
```

Utility methods I

There are a number of methods from the standard library that can very handy when dealing real world scenarios we will introduce a bunch of the more important.

`range(init, end, step)`

```
range(3) # 0, 1, 2
range(1, 3) # 1, 2
range(3, 1, -1) # 3, 2
```

`enumerate(iterator)`

```
for index, value in enumerate(['a', 'b']):
    print(index, value)

# 0, a 1, b
```

Utility methods II

`zip(i1, i2)`

```
a =[1, 2, 3]
b =['a', 'b', 'c']
for el_a, el_b in zip(a, b):
    print(el_a, el_b)

# 1, a 2, b 3, c
```

`dir(object)`

```
a =[1, 2, 3]
dir(a) # append, clear, copy, count, extend, index, insert, pop
                                    , remove, reverse, sort...
```

Exercise

1. Change the previous method so that the interpreter knows how to represent the object (toString())
2. Create a function that given a list of integers returns a list of the same integers powered to a number given by parameter. (([1, 2], 2) - [1, 4])
3. Create a function that given a string and a character counts the length of each substring divided by the character (('Hello world!', ' ') - [5, 6]).

Extras

Notebook

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

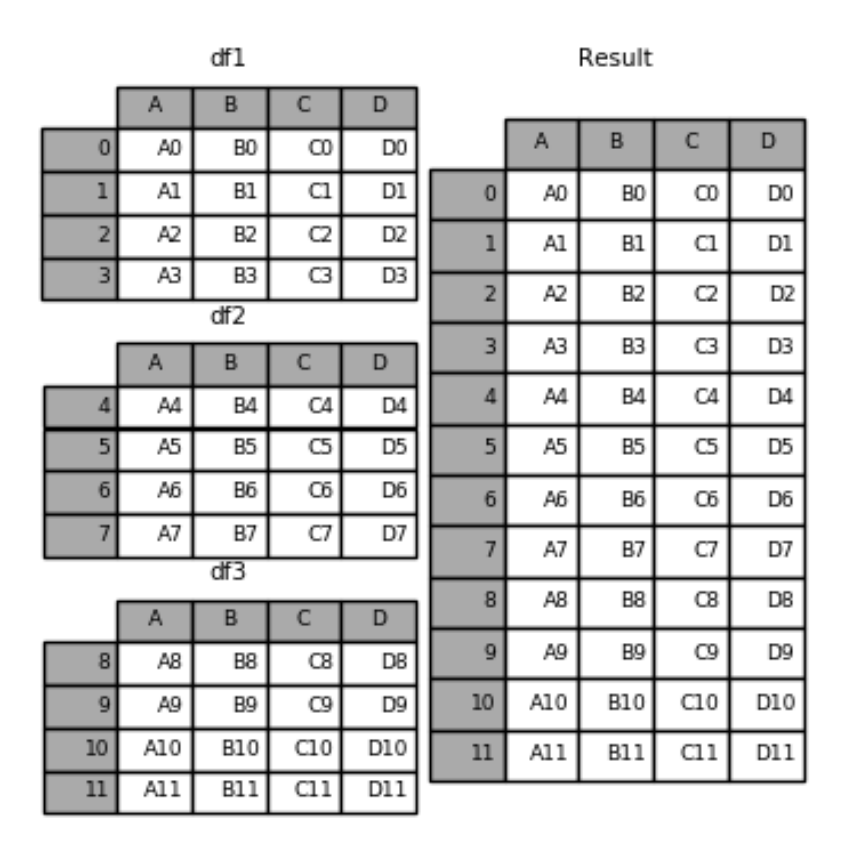
Sometimes we encounter missing values in our data, we have seen how to detect them, but to fill them with a imputation technique we can use the method `fillna`

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`.



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

Introduction to MatplotLib

Matplotplib

Matplotlib is a plotting library that will allow us to make graphical representations of data. Matplotlib is a huge library but we will focus on pyplot.

Matplotlib 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 matplotlib
```

The standard import of matplotlib is the following.

```
import matplotlib.pyplot as plt
```

Figure

A figure is where each plot reside, we may do several plots inside the same figure. Plots by default will be painted in the last figure generated.

```
plt.figure()
```

Subplots can be generated inside the same figure by calling `add_subplot` over the object. This methods need the number of columns and rows and the position of the subfigure.

```
fig =plt.figure()
fig.add_subplot(2, 2, 1) # 2 rows, 2 cols, first element
fig.add_subplot(2, 2, 4) # 2 rows, 2 cols, last element
```

Axis, title and ticks I

When we create a plot we get an axis object, this object gives us the possibility of putting labels or changing the ticks of each axis (x, y, ...) and much more things.

```
fig =plt.figure()
ax =fig.add_subplot(1, 1, 1)
```

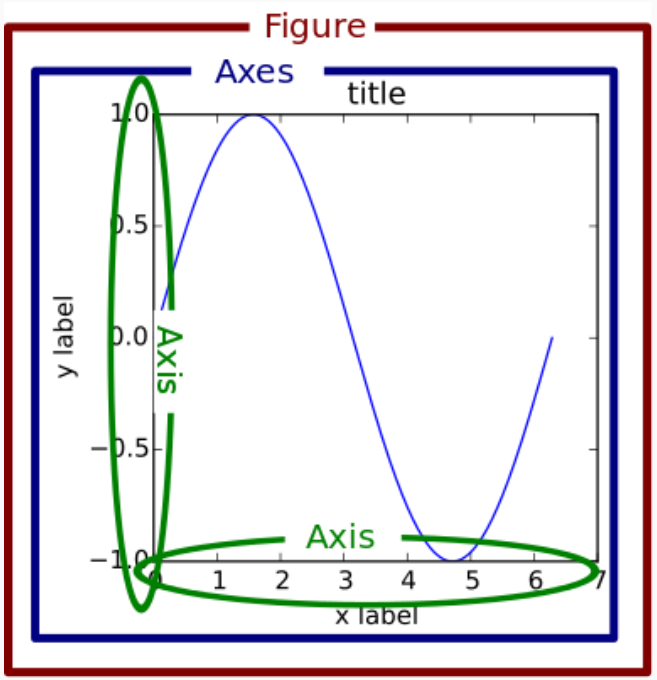
Ticks are the actual points that will be painted in the graph, each tick will have an associated label.

```
ax.set_xticks([-2, -1, 0, 1, 2])
ax.set_xticklabels(['-std2', '-std1', 'mean', 'std1', 'std2'])
```

Axis, title and ticks II

Some of the things that we can customize in the axis are the following.

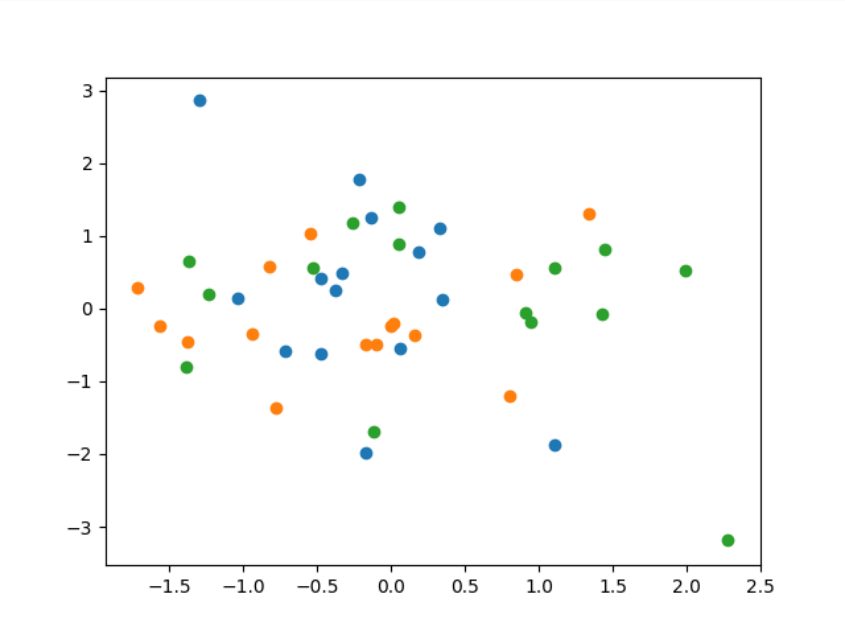
1. **xTicks** `a.set_xticks`
2. **xTicksLabels** `a.set_xtickslabels`
3. **yTicks** `a.set_yticks`
4. **yTicksLabels** `a.set_ytickslabels`
5. **Title** `a.set_title`



Scatter

A scatter graph is the one produced by single points, you need `x` and `y` coordinates in order to define a point, in matplotlib the method `plt.scatter` is the one used to perform this kind of graphs.

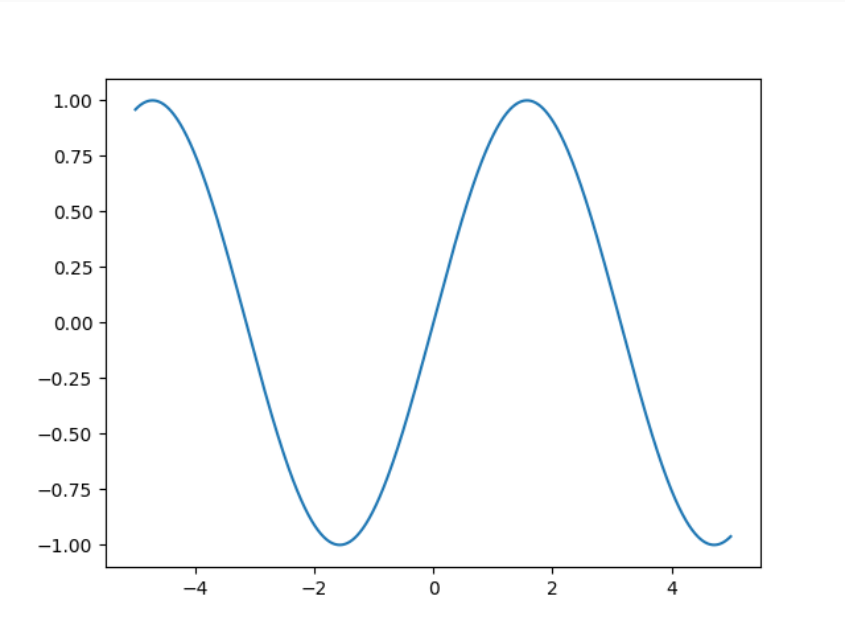
```
plt.scatter(
    np.random.normal(size=10), np.random.normal(size=10)
)
```



Line

Line plots are the ones produced by a function, you need to define x, y [and z]. The plotting is done with `plt.plot`.

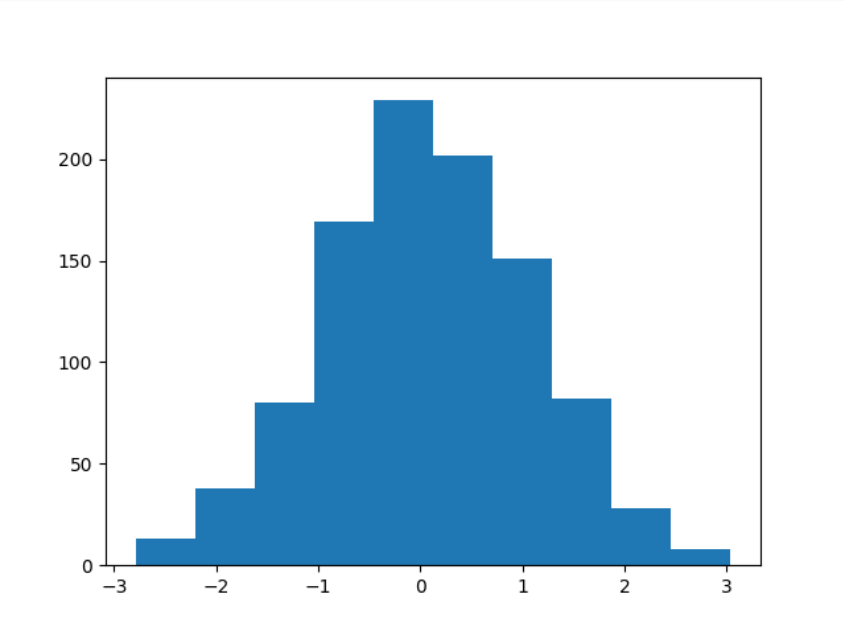
```
plt.plot(
    np.arange(-5, 6, 0.01), np.sin(np.arange(-5, 6, 0.01))
)
```



Histogram

A histogram is a graph representing the frequencies of data. Maplotlib only needs an array of vectors for calculating the histogram. The method `plt.hist` will produce this kind of graphs.

```
plt.hist(
    np.random.normal(size=100)
)
```



Pie chart

A pie chart is a special kind of chart that uses a circular representation. A vector is need by matplotlib to do this representation. Kepp in mind that values will be normalized. The method is `plt.pie`

```
plt.pie(
    [0.2, 0.3, 0.5]
)
```

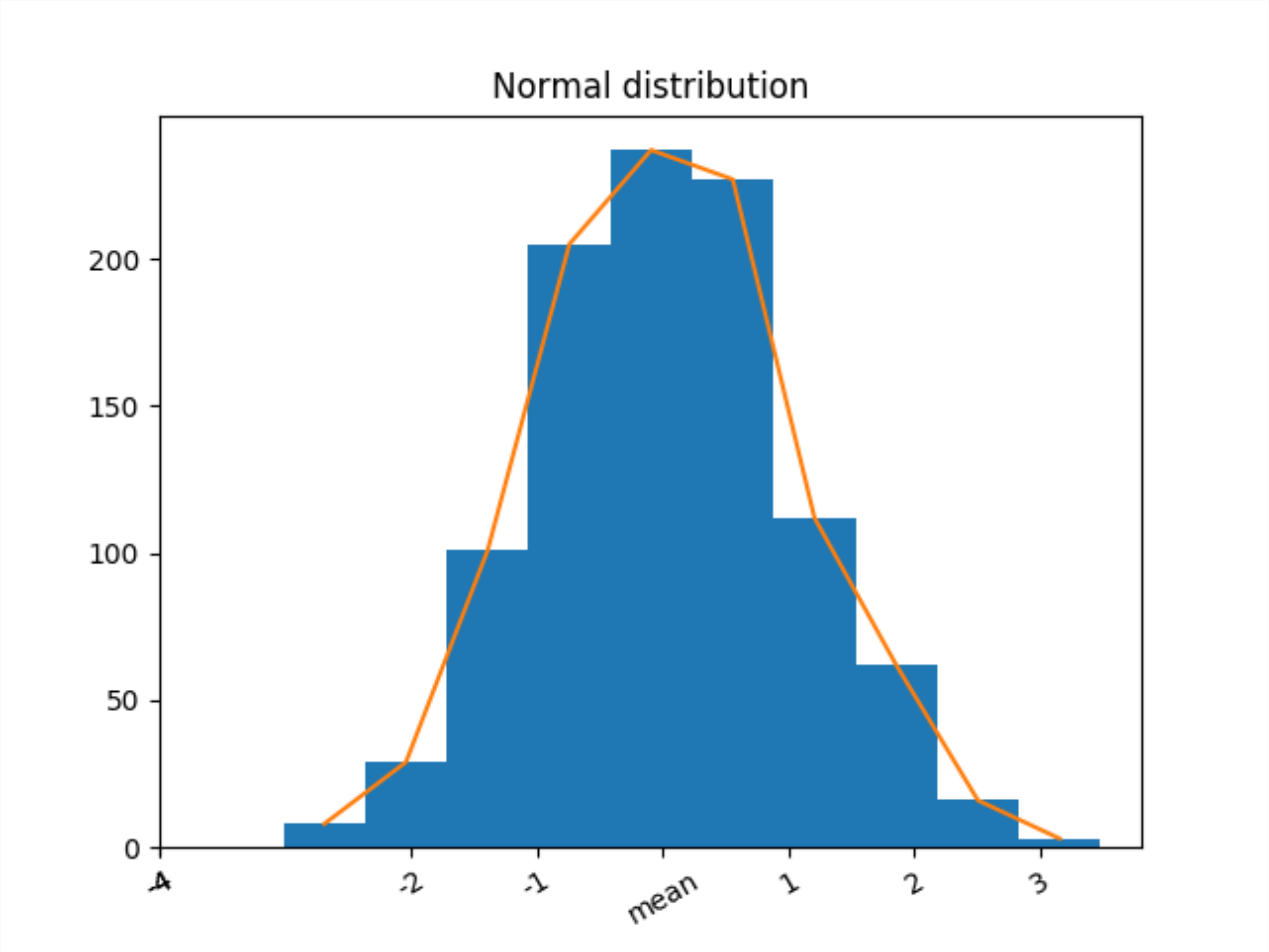
The plot method

All these exposed methods have a common interface to define some kind of visual properties.

1. **legend** A text that will be displayed gibing name to the plots in the figure.
2. **style** In scatter and line plots you can define the type of line or point with a third parameter indicating color and type of point in a string more info here

Exercise

Replicate this graph



Saving graphs

This graphs can be saved programatically in images by the method `plt.savefig`.

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

Pandas Integration

Pandas allows us to plot directly from a series by using the `plot` method together with the kind parameter. So for drawing a histogram for example we will do.

```
df =pd.Series([0.1, 0.2, 0.3, 0.5]).plot(kind='hist')
```

Jupyter Integration

Jupyter have a very nice integrations which lets you plot the graphs inline, to do so at the beginning of the notebook you must write `%matplotlib inline`

Useful to know libraries

There are some libraries built on top of matplotlib that give us special functionalities. Two of those are.

1. **Seaborn**
2. **Pyplot**

Extras

Notebook

Introduction to SkLearn interface

Matplotplib

Sklearn is the Swiss knife of machine learning, it comes with dozens of models out of the box and a huge community.

Sklearn 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 sklearn
```

Models in sklearn are imported separately as for example.

```
from sklearn.ensemble import RandomForestClassifier
```

Types of models

Inside of sklearn we will find different types of models. I will just introduce the high level API of them.

- **Supervised models** Models to perform predictions.
- **Unsupervised models** Models to group data automatically.
- **Transformation models** Models to perform transformations in the data.

Supervised models

This kind of models are the most intuitive ones. You train them with data and expected outputs and later it will predict outputs for unseen data. To train the algorithm we call the `fit` method and to predict with it we call the `predict` function.

```
from sklearn.ensemble import RandomForestClassifier

clf =RandomForestClassifier().fit(X, y)
clf.predict(X)
```

Unupervised models

Other type of models, in this case it will not predict but find groups of similar elements inside data. To train the algorithm we call the `fit` method and to get the group of an unseen element we call the `predict` method.

```
from sklearn.cluster import KMeans

clf =KMeans().fit(X)
clf.predict(X)
```

Transformation models

This last kind of models transform our data. For example for dimensionality reduction tasks or normalization tasks. Their main method is `fit_tranform`

```
from sklearn.preprocessing import MinMaxScaler

transformed_data =MinMaxScaler().fit_transform(X)
```

Extras

Notebook