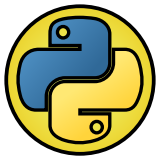
Python for Science

A minimal guide to do large things

Contents



Author(s) : *D. André*
Contributor(s) : *A. Boulle*

January 25, 2020

This document is placed under the creative commons BY-SA license

Part 1

Beginning with python

1 Introduction

1.1 Four reasons for learning and using python

A common problem for scientist is data analysis. These data generally come from experimental equipment or numerical calculations. Spreadsheet software are widely used to perform this task. Spreadsheet software are comfortable for users, they give smart graphical interfaces with smart features. However the number of functionality is limited and, for advanced usages, users are not able to find a function that matches their expectations. You probably have experienced this limitation! To face this problem, a common solution is to extend spreadsheet software with *macros*. Macros are pieces of code written by users in order to extend software.

In my opinion this is not an optimal solution. Learning macros is like learning programming languages... plus additional things such as specific interfaces with the specific software. So, if you are ready to spend time for learning macros, my advice is to spend this time for learning *a real* generalist programming language... such as Python!

I admit that the time investment is quite larger than time for learning macro languages but the results are exponential. You will be amazed by the power of the Python language. With only fundamental backgrounds you can do a huge number of advanced things: data analysis, scientific computations, web coding, programming games and so on... The sole limitation is your imagination!

Okay, but why learning specifically the Python language? Several languages exist: C++, matlab, C, Fortran, Java, Haskell, Bash, VB.net... The reasons are: Python is **easy**, Python is **free**, Python is **multi-platform** and Python is **widely used** in the scientific community.

To highlight its simplicity, take a look at this piece of C++ code. This code snippet simply displays "`hello world`".

```
#include<iostream>
int main(void)
{
  std::cout << "hello world" << std::endl;
  return 0;
}
```

Now, look at the Python version of this program.

```
print ("hello world")
```

You can see that only one line is used in the Python version while six lines of obscure code are used in the C++ version. In fact, Python was specifically designed to be simple, intuitive and elegant.

One other reason for learning Python is that Python is free. *Free* means that you can use Python without pay anything. It is nice but *free* means more than that. You have also a free access to the Python code (Python is coded in C): you can modify it and copy it without any restriction. You are probably not directly concerned by these specific advanced usages but it's really important that other people can do such things for you. For example, such people have devel-

oped the python environments *Anaconda* and *python(x,y)*. These environments offer complete, documented, smart and easily instalable python workbenches.

Python is also *multi-platform*. It means that a Python code can be interpreted with the same behavior on the three main operating systems: GNU/Linux, Mac OSx and Windows.

For all these reasons Python is really popular, especially in the scientific community. Using a popular and free language is really comfortable because it promotes the sharing of sources. People who want to share their developments generally package them into *modules*. Modules are a very smart method to import in your own Python environment third party libraries. For example, the next listing shows how to use the `smtplib` module to send automatically an email in only six lines of code! Of course, don't use it for spamming your friends :)

```
import smtplib

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login("YOUR_GMAIL_ADDRESS", "YOUR_PASSWORD")

server.sendmail("YOUR_GMAIL_ADDRESS", "THE_EMAIL_ADDRESS_TO_SPAM", "Hello")
server.quit()
```

This second example, shows how to use the nice `fbchat` module for using the facebook chat without any web browser.

```
from fbchat import Client
from fbchat.models import *

client = Client('YOUR_FB_LOGIN', 'YOUR_FB_PASSWORD')
print('Own id: {}'.format(client.uid))
client.send(Message(text='Hi me!'), thread_id=client.uid, thread_type=ThreadType.USER)
client.logout()
```

Please note that these examples are not very useful and I not encourage yourself to use non-free services such as Gmail or Facebook. But imagine that you can couple this last code snippet with artificial intelligence modules. You are able to create smart bots for detecting terrorism activities in this social network! Python allows to combine easily all the main numerical technologies. Feel free to play with them like a wizard!

For example, I use Python for running a predefined list of computational test cases (also called *unit tests*) of the GranOO software. The computational time and results are monitored and a daily report is sent automatically by e-mail to the GranOO developers in order to check the non-regressive aspects of the new versions. I have used also Python for developing an android application named *pyrfda* able to measure the young's modulus of material with a smartphone.

Today, more than 10,000 packages and modules are available in the official Python repository named *PyPI*. If you want to code something in Python, a preliminary step is to search if someone has already coding this task. For instance, if you want to create beautiful interactive charts, you can use the *matplotlib* package, if you want to apply a Fast Fourier Transform (FFT) to your data for spectral analysis, you can use the *scipy* package or if you expect linear algebra computations, you can use the *numpy* package.

These three last packages (matplotib, numpy and scipy) are such as swiss knives for scientists. They will be detailed later in this document. But at this time, you need to learn how to install python on your computer and the fundamentals of this incredible language.

1.2 Installing Python

Note that several methods exist for installing Python. The methods proposed here are simple and use pre-packaged version of Python.

Today, two versions of Python are maintained. The 2.7 and 3.x versions. Note that 2.x and 3.x versions are not compatible. For beginners there is only few differences between these two versions. My advice is use Python 3.x because the 2.7 version will be gradually abandoned. **This document will treat only of Python 3.x versions**.

For windows, you can use the *anaconda* Python distribution. You will find at this address www.anaconda.com/download/ the required installers. To avoid any further problems, choose an installation path without any space or special characters. To check your install, run the *anaconda prompt* and type `idle` inside it. If the install is right, a graphical interface shall open.

For MacOsx, you can use also the *anaconda* Python distribution. You will find at this address www.anaconda.com/download/ the required installers. To avoid any further problems, choose an installation path without any space or special characters. To check your install, run the `idle3` command in a terminal. If the install is right, a graphical interface shall open.

For GNU/Linux, Python is already packaged. On a debian based system you can install python3 with scientific tools thanks to the following terminal command.

```
sudo apt-get install python3 ipython3 ipython3-notebook idle3 python3-pyqt5 python3-pip \
python3-numpy python3-scipy python3-matplotlib
```

Now, you can type `idle3` in a terminal. If the install is right, a graphical interface shall open.

Independently of you operating system, if you run the *IDLE* program, you should see something like this (see figure 1).

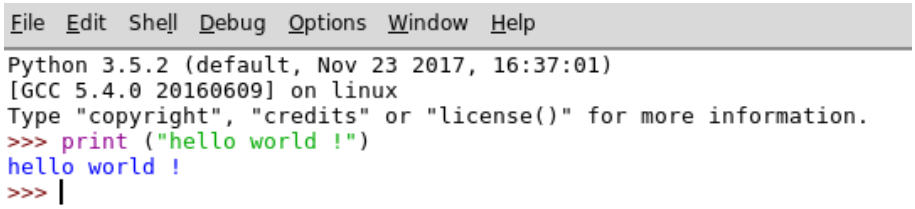


Figure 1: The interactive Python interpreter shown by *IDLE* with the "hello world" example

Welcome to the magic world of Python! Now, you can start playing with the interactive Python interpreter. The Python interpreter interprets your command each time you press the "enter" key. For example, you can type the following line in the interpreter

```
>>> print("Hello world !")
```

and if you press the "enter" key, the answer of the interpreter will be:

```
Hello world !
```

Of course, you can do basic mathematical operations such as:

```
>>> 5+4   # addition
9
>>> 5-4   # subtraction
1
```

```
>>> 5/4   # division
1.25
>>> 5//4  # integer division
1
>>> 5*4   # multiplication
20
>>> 5**4  # power
625
```

> ℹ *With Python, the "#" character is used for comments. You can use comments for describing your code. Of course, a comment is not interpreted by Python parsers.*

You can use the interpreter as a powerful pocket calculator thanks to variables. For instance, the following code add the content of the variable `a` to the content of the variable `b` and displays the results.

```
>>> a = 4
>>> b = 5
>>> a+b
9
```

In Python the equal sign "=" is named *affectation*.

> ℹ *If your are familiar with C or C++, note that the affectation mechanism is not equivalent to variable creation in C or C++. When you type `a=4`, python allocate the memory space needed for a new integer that contains the `'4'` value. The variable "`a`" simply points on the address of `'4'`. Python uses a garbage collector, so you don't need to manage memory by yourself.*

If you use the equal sign "=" to affect values to variables in your line code, you can note that the output is silent. If not, the interpreter returns the result of your command. The >>> characters means that the interpreter is waiting for your command. The interpreter answers are not prefixed by any character.

The interpreter is a good tool for quickly testing such little things. However, if you want to build complex programs, you should write *script* files. A script file is simply a text file that contains a sequence of Python commands. With *IDLE*, you can create a new script file by clicking "`File > New File`". The whole content of the script file will be interpreted by pressing the "F5" key. For instance, you can create the following script file, that contains only 2 lines of code.

```
msg = "Hello world\n"
print (msg*5)
```

Now, if you press the "F5" key to execute your script. The interactive interpreter should display

```
>>>
============================ RESTART: test.py ============================
Hello world
Hello world
Hello world
Hello world
Hello world
```

If you place the python interpreter window contiguously to the script windows, you should see something like figure 2.

IDLE is an Integrated Development Environment (IDE) devoted to the Python language. I recommend to start with this editor because it is simple and powerful. When you will be more

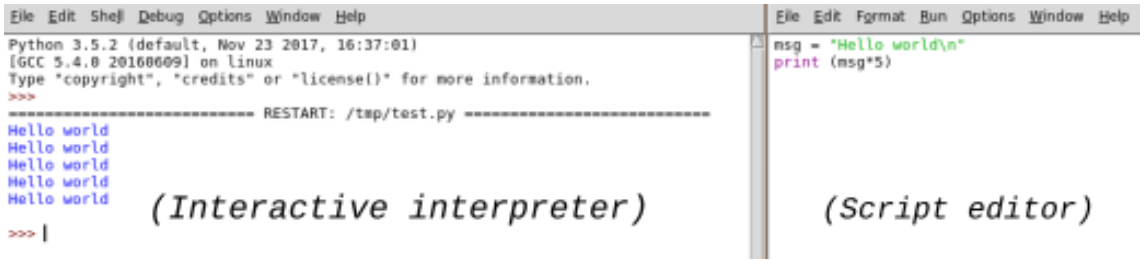


Figure 2: The *IDLE* python interpreter and script editor

familiar with Python, you can use more advanced IDE such as *idleX*, *Spyder*, *Eclipse*, *Emacs* or *Vim*. You can use also simple code editors such as *Notepad++*, *Kate*, *Geany*... All these tools have specific advantages. Feel free to document yourself about these tools and choose the most adapted to your usage. Note that, in the scientific and education community, the *IPython notebook* is one of the most popular.

At this stage, you must have a ready-to-use python environment. Now, you will learn the basics of the Python Language.

2 Language basics

In this section, you will learn the basics of the Python language. Note that with only basic knowledge, you are able to build complex programs!

2.1 Built-in types

Python is a *dynamically strongly typed* language. It means that all variables own a unique type. In the Python terminology, a *type* is also named a *class*. The class concept, that comes from the Object Oriented Programming (POO), is an advanced programming technique. We will learn the basics of the class and POO concepts later in this document.

If you build a new variable, you can access to the type of a variable thanks to the the *built-in* function `type()`.

> ℹ *A built-in function is a function made by Python developers. So, you can use built-in functions anywhere in your Python program. Python defines also built-in variables and built-in types.*

The following lines show the usage of the built-in `type()` function.

```
>>> a=1
>>> type(a)
<class 'int'>

>>> b=3.14
>>> type(b)
<class 'float'>
```

With this example, the type of the variable `a` is `int` and the type of the variable `b` type is `float`. Of course `int` means integer and `float` means floating point number.

ℹ *Be aware, floating points are not a perfect representation of numbers. For instance:*

```
>>> ((0.7+0.1)*10)
7.999999999999999
```

The result gives `7.999999999999999`*. If you need more precision, you can use the* `decimal` *module that comes from the standard Python library.*

Python use *dynamic* typing. It means that a variable type can be modified during the execution of a program. Look at the following code.

```
>>> a=1
>>> type(a)
<class 'int'>

>>> a=3.14
>>> type(a)
<class 'float'>
```

The `a` variable changes from `int` to `float`. This behavior is not allowed with static typing languages such as C++.

You can "force" a type by invoking explicitly type constructors. Constructors, that came from POO, are special functions that build new instances of a type (in fact this is not a type, this is a class). Let's see an example that highlights this concept:

```
>>> b=int(3.14)
>>> type(b)
<class 'int'>
>>> b
3
```

In this example, the line `int(3.14)` constructs a new instance of the `int` type from the `3.14` floating point number. The constructor of `int` is explicitly given, so Python convert `3.14` to an integer. Finally, the result is affected to the `b` variable. So, the `b` variable is an integer that points on the `3` value.

In some cases, Python does implicit conversion for you. The following example, highlights this behavior:

```
>>> a=1
>>> b=3.14
>>> type(a+b)
<class 'float'>
```

In this example, you can note that an addition between a float and an integer gives a float! You must be aware of this behavior, it is powerful but also dangerous if you do not expect the right returned type.

Of course, you can "force" the type by invoking class constructors such as:

```
>>> a=1
>>> b=3.14
>>> type(int(a+b))
<class 'int'>
```

The `int` and `float` types are called *numeric type*. Common mathematical operations such as addition, subtraction and so on.... are available with numeric types.

> ℹ *For those familiar with C or C++, you probably want to know the memory address and the memory size of variables. You can use the built-in function `id()` that returns an address and the `getsizeof()` function that returns a memory size. Note that the `getsizeof()` function is architecture dependent and comes from the `sys` module. If you play with these functions you probably will be surprised. Python and C or C++ are really different!*

Another useful built-in type is *string*. Strings (a.k.a `str`) are character chains able to display word, sentences and so on. You must declare a string between simple quote or double quotes. For example:

```
>>> a = "Hello "  # str construction with double quote
>>> b = 'world !' # str construction with simple quote
>>> a + b
'Hello world !'
>>> 10*a
'Hello Hello Hello Hello Hello Hello Hello Hello Hello Hello '
>>> a + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

You can note that `str` type supports some mathematical operators such as addition "+", that concatenate strings, or multiplication with an integer "*" for replication. However, as you can see, addition between a `str` and an `int` is not allowed!

You can access to a substring with the bracket operator "`[i]`", where `i` is the index of the character in the string chain. For instance, the following lines extract the first character of the string.

```
>>> a = "Hello world"
>>> a[0]
'H'
```

You can access to a substring thanks to *slice*. Slice defines sublist thanks to the `[n:m]` syntax as follows:

```
>>> a = "Hello world"
>>> a[0:5]
'Hello'
>>> a[6:11]
'world'
```

Note that you can pass negative index to the bracket operator. You just have to backward count. For instance the `-1` index returns the last character.

```
>>> a = "Hello world"
>>> a[-1]
'd'
```

You can use negative index with slice as follows:

```
>>> a = "Hello world"
>>> a[6:-1]
'worl'
```

If you want to slice from an index to the end of string, you can use `[n:]` as follow

```
>>> a = "Hello world"
>>> a[6:]
'world'
```

Or, you can start from the beginning of string with `[:m]` as follows

```
>>> a = "Hello world"
>>> a[:5]
'Hello'
```

You can also gives a step for slicing a string with `[n:m:step]` syntax such as:

```
>>> a = "Hello world"
>>> a[0:11:2]
'Hlowrd'
```

So, you can revert a string by passing a negative step such as:

```
>>> a = "Hello world"
>>> a[::-1]
'dlrow olleH'
```

The Python string type is really powerful. A large number of functions and *methods* are available.

> ℹ *A method is a special kind of function that acts on an instance of a class (an object). To invoke a method on an object, the syntax is `object.method()`, where the "." indicates that the method "`method`" acts on the object "`object`". Note that, as classical functions, some arguments can be passed to methods: `object.method(arg1, arg2)`.*

To get the whole list of methods associated to a given class, you can invoke the built-in `dir()` function such as:

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__'
    , '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '
    __init__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__'
    , '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__
    ', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', '
    endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', '
    isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', '
    istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', '
    rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith
    ', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Suppose that the `lower` method triggers your curiosity and you are looking for additional information about this method. You can get specific documentation with the `help()` built-in function.

```
>>> help(str.lower)
Help on method_descriptor:

lower(...)
    S.lower() -> str

    Return a copy of the string S converted to lowercase.
```

The documentation tell us that the `lower` method converts strings in lowercase. We can use the `lower` method as it follows:

```
>>> a = "HELLO"
>>> a.lower()
'hello'
```

If you want to learn more about strings (or other built-in types), the online official Python documentation available at https://www.python.org/ gives exhaustive lists of features highlighted by pedagogical examples.

In fact, string is a specific application of the *list* class. Lists are the default containers of Python. Containers are such arrays that contains *heterogeneous* objects. Heterogeneous means that different kind of objects can be indexed by a list. Let's see an example:

```
>>> li = [3.14, 1, "Hello"]
>>> type(li)
<class 'list'>
>>> li
[3.14, 1, 'Hello']
```

As for strings, an element of the list can be accessed with the bracket operator "`[i]`".

```
>>> li = [3.14, 1, "Hello"]
>>> li[0]
3.14
```

You can change an element of the list, for instance:

```
>>> li = [3.14, 1, "Hello"]
>>> li[0] = 6.28
[6.28, 1, 'Hello']
```

The size of a list is returned by the built-in `len()` function

```
>>> li = [3.14, 1, "Hello"]
>>> len(li)
3
```

You can dynamically add an element to a list thanks to the `append()` method of the list class:

```
>>> li = [3.14, 1, "Hello"]
>>> li.append(True)
>>> li
[3.14, 1, 'Hello', True]
```

Here we append a Boolean variable equal to `True`. A Boolean can take only two values: `True` or `False`. A good trick is to use the keyword `in` to check if an element is in a list:

```
>>> li = [3.14, 1, "Hello"]
>>> "Hello" in li
True
```

An item can be removed from a list thanks to the `remove` method of the list class.

```
>>> li = [3.14, 1, "Hello"]
>>> li.remove(3.14)
>>> li
[1, 'Hello']
```

Python uses also `tuple` class. Tuples are non-modifiable lists. They are defined with parenthesis `()` instead of bracket `[]` such as:

```
>>> t = (3.14, 1, "Hello")
>>> t[0]
3.14
>>> t[0] = "test"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

In some cases, tuples can be advantageous used instead of lists because tuples are *faster* than lists and for other reasons...

> ℹ *Python defines mutable and immutable types. A mutable type means that the type is modifiable. An immutable type means that the type is not modifiable. The common types `int`, `float`, `str` and `tuple` are immutable ones.*

A last common built-in type is dictionaries (a.k.a `dict`). A dictionary is a special kind of container that associates *keys* with *values*. For instance, you can use a dictionaries as en entry for a telephone book:

```
>>> entry = {'name':'john doe', 'phone number':55555555}
>>> entry['name']
'john doe'
>>> entry['phone number']
55555555
>>> entry['address']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'address'
>>>
```

Here, two string keywords are used: `'name'` and `'phone number'`. As for lists and tuples, an item can be accessed with the bracket operator `[keyword]`. However, you need to pass the keyword instead of the index number in the bracket operator. If you specify an unknown key, it raises an error (to be more precise, it triggers an *exception*). The list of keywords and the list of associated values can be accessed with the `keys()`, `values()` and `items()` methods:

```
>>> entry = {'name':'john doe', 'phone number':55555555}
>>> entry.keys()
dict_keys(['phone number', 'name'])
>>> entry.values()
dict_values([55555555, 'john doe'])
>>> entry.items()
dict_items([('phone number', 55555555), ('name', 'john doe')])
```

Note that dictionaries are mutable, you can change a value contained in a dictionary as:

```
>>> entry = {'name':'john doe', 'phone number':55555555}
>>> entry['phone number'] = 333333333 # changing phone number here
>>> entry
{'phone number': 333333333, 'name': 'john doe'}
```

You can use all *immutable* types as dictionary keywords.

This section gives a very quick overview of the main Python built-in types. If you want an exhaustive list of the Python built-in types with their features, you can refer to the official python3 documentation https://docs.python.org/3/library/stdtypes.html

2.2 Useful built-in functions

2.2.1 Displaying text with `print()`

The print function is one of the most used built-in functions. It allows nice outputs, with string concatenation, string conversion and so on. The following example highlights these features.

```
>>> a = 'Hello'
>>> b = 2
>>> c = 'you'
>>> print(a)
Hello
>>> print(a,b,c)
Hello 2 you
>>> print(a,b,c, sep=", ")
Hello, 2, you
>>> print(a,b,c, end="!\n")
Hello 2 you!
```

Note the usage of the special character "\n" that means *end-of-line*. Another trick here is the usage of named input arguments `sep` and `end`. Generally, named arguments are optional. If you take a look into the documentation of the `print()` function, these arguments are well documented:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

You can see here that the default values of the named arguments `sep` and `end` are respectively a white space and an end of line.

A good trick is to use the `format()` method that comes from the `str` built-in type. The following example shows how to display a real number with only two digit numbers thanks to this method:

```
>>> val = 1/3
>>> val
0.3333333333333333
>>> print ("val={:0.2f}".format(val))
val=0.33
```

2.2.2 Getting user input with `input()`

The `input()` built-in function make a pause in a program execution until the user presses the "enter" button. For instance:

```
>>> ans = input('please enter a word: ')
please enter a word: hello
```

```
>>> print ("your word is", ans)
your word is hello
```

In this example, a user types "`hello`" and presses enter. The entered string is stored in the `ans` variable. Finally, the `print()` function is used to display the user entry.

It looks strange to use the `input()` function inside the interactive terminal because you can assign directly the `ans` variable. Therefore, it makes sense to use it inside a Python script for building interactive programs. The following example asks user for a number and displays the square of this number.

```
ans = input('please enter a number: ')
num = float(ans)
print("the square of your number is", num**2)
```

If you execute this script (use "F5" with IDLE), the given output is :

```
please enter a number: 3
the square of your number is 9.0
```

2.3 User defined functions

Defining function is base of coding. Functions, in programming techniques, are closed to the function concept in mathematics. A function has a name, it takes none, one or several arguments (entries) and returns none, one or several values (outputs). Functions are generally used for *factorizing* codes. Factorizing avoid to rewrite several times the same code in different places of a program.

The following code defines a function named `hello` that takes no variable as argument and returns nothing.

```
>>> def hello():
...     print ("hello")
```

> ℹ *You can see that the `print ("hello")` line does not start from the beginning of line. In fact, the code is indented. A common usage is to use four white spaces as one indentation level. Line codes that use the same indentation level are such as code blocks. Python expect a new indentation level after the " :" semi-colon character.*

Now, we can use the `hello` function as:

```
>>> hello()
hello
```

Note that the `hello` function returns nothing... and nothing itself is the Python class `NoneType`! Python is really incredible.... ;)

```
>>> res = hello()
hello
>>> type(res)
<class 'NoneType'>
```

This other example defines a function named `power` that takes two variables as argument and returns one result:

```
>>> def power(base,exponent):
...     return (base**exponent)
```

You can use the `power` function as follows:

```
>>> power(2,3)
8
```

You can use also named arguments when you invoke functions:

```
>>> power(base=2,exponent=3)
8
```

Note that you can change the order with named argument as follows:

```
>>> power(exponent=3, base=2)
8
```

Multiple variable return is alos available in Python. The following code snippet defines a function named `euclidean_div` that takes two variables as argument and returns two values: the quotient and the remainder. Note the usage of comma "," for separating the returned values.

```
>>> def euclidean_div(a,b):
...     quotient = a//b
...     reminder = a%b
...     return quotient, reminder
```

We can invoke this function as:

```
>>> euclidean_div(13, 4)
(3, 1)
```

You can note that the function returns a tuple. You can use multiple variable assignment to catch the returned values:

```
>>> a,b = euclidean_div(13, 4)
>>> print ("quotient=", a, "reminder=", b)
quotient= 3 reminder= 1
```

An advanced usage of input arguments is the usage of `*args` and `**kwargs`. The `*args` argument is a tuple that lists all the parameters sent by user. The following example highlights this feature.

```
>>> def test_args(*args):
...     print (args)
...
>>> test_args(1, 2, "hello", "world")
(1, 2, 'hello', 'world')
```

The second one `**kwargs` is a dictionary that contains the list of named arguments. The following code highlights this feature:

```
>>> def test_kwargs(**kwargs):
...     print (kwargs)
...
>>> test_kwargs(i=1, j=2, k="hello", l="world")
{'l': 'world', 'j': 2, 'i': 1, 'k': 'hello'}
```

> ℹ *Python dictionaries are not ordered. That's why the above result is not in the same order that function arguments.*

If you specify only `**kwargs`, you are not able to use non named arguments. The following code snippet gives an error because the first argument has no name:

```
>>> test_kwargs(1, j=2, k="hello", l="world")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test_kwargs() takes 0 positional arguments but 1 was given
```

An advanced usage is to combine both `*args` and `**kwargs`. By the way, your are able to manage named and non-named arguments. This is a powerful feature often used in python libraries. The following code shows how to combine both arguments:

```
>>> def test(*args, **kwargs):
...   print(args, kwargs)
...
>>> test(1, 3.14, j=2, k="hello", l="world")
(1, 3.14) {'k': 'hello', 'j': 2, 'l': 'world'}
```

2.4 Loops

A very common operation with Python is to loop over *iterable* objects. Iterable objects come from classes that define the reserved `__iter__()` method. To be simple, lists and tuples are iterable. Iterating over a list or a tuple is managed by the keywords `for` and `in`. The following code shows how to build a new list and iterate over it:

```
>>> my_list = [1, 2, "hello", 3.14]
>>> for i in my_list:
...     print(i)
...
1
2
hello
3.14
```

> ℹ *As for functions, a `for` loop must define a new code block. Remember that a block begin with the " :" character following by an increment in the indentation level. All line codes that belong to a given block must have the same (or upper) indentation level.*

The built-in function `range(start,stop,step)` creates an iterable list of integer. This function is commonly used to iterate over a list of integers. The following code shows some basic usages of this function. Note the usage of the list constructor `list(...)`.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

It is possible to iterate over a range with:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Python supports list comprehensions. This is a very powerful feature able to build complex list structure on-the-fly. For example, the following code build the list of the squared ten first numbers:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehensions are a very amazing feature for manipulating lists. It allows conditional instructions. The following code build the list of the first ten squared numbers only if the related numbers are pair.

```
>>> [x**2 for x in range(10) if x%2==0]
[0, 4, 16, 36, 64]
```

Another kind of loop is the `while` loop. A while loop is simply defines as this:

```
>>> i = 0
>>> while i < 5:
...     print(i)
...     i += 1
...
0
1
2
3
4
```

You can skip a `while` loops and `for` loops with the `break` keyword as:

```
>>> i = 0
>>> while True:
...     print(i)
...     i += 1
...     if i > 5:
...         break
...
0
1
2
3
4
5
```

2.5 Conditions

As for loops, conditions are defined through code blocks. The keywords `if`, `elif`, and `else` can be used to create conditional instructions. For example, the following code block tests if the `x` variable defined by a user is higher than ten:

```
>>>> x = int(input("give me a number: "))
give me a number: 13
>>> if x > 10:
...     print ("higher that 10")
...
higher that 10
```

The following code snippet shows how to use the `else` keyword:

```
>>>> x = int(input("give me a number: "))
give me a number: 7
>>> if x > 10:
...     print ("higher that 10")
... else:
...     print ("lower that 10")
...
lower that 10
```

The `elif` keyword can be used for adding conditions, such as:

```
>>> x = int(input("give me a number: "))
give me a number: 7
>>> if x > 10:
...     print ("higher that 10")
... elif x > 5:
...     print ("between 5 and 10")
... else:
...     print ("lower than 5")
...
between 5 and 10
```

The following operators are available in conditional declaration:

- "<" strictly inferior
- ">" strictly superior
- "<=" inferior or equal
- ">=" superior or equal
- "==" equal
- "!=" different

In addition, you can use the keywords `in`, `is`, `not` and `or`. It allows to write conditions in a very intuitive way:

```
>>> msg = "Hello world"
>>> if "Hello" in msg:
...    print("your message is 'Hello something'")
...
your message is 'Hello something'
>>> a = 5
>>> if a is 5:
...    print ("the 'a' variable is 5")
...
the 'a' variable is 5
>>> if type(a) is not int:
...    print("'a' is not an integer")
... else:
...    print("'a' is an integer")
...
'a' is an integer
```

2.6 Importing modules

At this time, you have seen only built-in Python features. Modules can be used to import third-party features in your Python environment. For example, the trigonometric function *cos(x)* or *sin(x)* are not implemented in the built-in Python environment. If you need these trigonometric functions, you can use the `math` module that comes from the standard Python library. Importing modules are given thanks to the `import` keyword as it follows:

```
>>> import math
>>> math.cos(0)
1.0
```

If you want to use a module function, class or variable, you need to prefix their names by the name of the module. To keep life easier, you can associate alias with a module as it follows:

```
>>> import math as m
>>> m.cos(0)
1.0
>>> m.pi
3.141592653589793
```

If you are really tired, you can use the special character `*`. It avoids any prefix:

```
>>> from math import *
>>> cos(0)
1.0
>>> pi
3.141592653589793
```

> ℹ *The above code imports the whole content of a module in the current namespace. Be careful, you run the risk of doing name collision! To be brief, this is not safe.*

2.7 Creating your own modules

Creating modules is a smart way to share a part of your code between different applications. Supposes that you create a Python script file named `my_algo.py` that contains:

```
def run(x):
    print (x * "RUN !!!")
```
my_algo.py

If you create a new script in the same directory, you can use the `my_algo.py` script as a module. For example, suppose that you create a new Python script file named `test.py`. Your directory tree must look like:

```
├── test.py
└── my_algo.py
```

Suppose that the `test.py` script contains:

```
import my_algo
my_algo.run(10)
```
test.py

Because both files are in the same directory, you are able to import the content of `my_algo.py` with the `import` keyword. If you run the `test.py` script, it gives:

```
RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!RUN !!!
```

Now, if you want to place `test.py` in another directory from `my_algo.py`, you must help python to locate your module. To do such thing, you can use the `imp` module as:

```
import imp
my_algo = imp.load_source('my_algo', '/path/to/my_algo.py')
my_algo.run_algo(10)
```
test.py

Another way to do that is to edit the environment path as:

```
import sys
sys.path.insert(0, '/path/to/')

import my_algo
my_algo.run_algo(10)
```

You can also split a module in several files. This kind of module are named *package*. For instance, suppose that you want to split the `my_algo` module in three files: `algo1.py`, `algo2.py` and `algo3.py`. To do that, you must place this three files in a directory named `my_algo`. Your package gives the following file tree:

```
test.py
my_algo
    algo1.py
    algo2.py
    algo3.py
```

If the contents of the `algo1.py`, `algo2.py` and `algo3.py` files are respectively:

```
def run(x):
    print (x*"RUN1 ")
```
algo1.py

```
def run(x):
    print (x*"RUN2 ")
```
algo2.py

```
def run(x):
    print (x*"RUN3 ")
```
algo3.py

Now, you can import each module in the package with the following syntax: "`import package.module`". The following example imports the three modules inside the `test.py` script environment and executes the related `run(...)` function of each module:

```
import my_algo.algo1 as algo1
import my_algo.algo2 as algo2
import my_algo.algo3 as algo3

algo1.run(10)
algo2.run(10)
algo3.run(10)
```
test.py

if you run the `test.py` script file, it gives the following output:

```
RUN1 RUN1 RUN1 RUN1 RUN1 RUN1 RUN1 RUN1 RUN1 RUN1
RUN2 RUN2 RUN2 RUN2 RUN2 RUN2 RUN2 RUN2 RUN2 RUN2
RUN3 RUN3 RUN3 RUN3 RUN3 RUN3 RUN3 RUN3 RUN3 RUN3
```

3 Data analysis

A common usage of Python for scientists is data analysis. Data generally come from experimental apparatus or numerical computations and they are commonly embedded into text files.

3.1 Reading text files

Several data file formats exist. Data file formats are often a problem because no standard really exists. We can cite the standard HDF5 file format able to deal with very large data or the CSV format commonly used for "little" data. For its simplicity, the CSV (Comma Separated Value) files are very popular but this is not a really standardized one. In practice, users have to adapt their file readers to each specific case. In this section, we will see how to read a non-standard data text file.

The following block shows the content of a non standard data text file. You can download this file at: http://www.unilim.fr/pages_perso/damien.andre/cours/python/data.txt. The first four lines that begin by "#" characters are comments and must be ignored. The lines after comments contain the wanted data. Data are simply separated by tabulations.

```
# column: 0, label: iteration
# column: 1, label: bond
# column: 2, label: force
# column: 3, label: position
0 0 1.2261431322e+05  -4.6449842528e-01
20  0 5.5783516921e+05  -4.6450242528e-01
40  0 5.2612365475e+05  -4.6450642528e-01
60  0 5.1126459531e+05  -4.6451042528e-01
80  0 5.0346137297e+05  -4.6451442528e-01
100 0 5.9211801170e+05  -4.6451842528e-01
120 0 8.8921281016e+05  -4.6452242528e-01
....
```
data.txt

Suppose that we want to read this file with Python: we want to store the "iteration" and "force" columns in two separated lists: `it` and `force`. The following code snippet does this job.

```
it     = []
force  = []

with open('data.txt', 'r') as f:
    for line in f:
        if not '#' in line:
           data = line.split()
           it.append(int(data[0]))
           force.append(float(data[2]))
```

Only eight lines! We can do smaller but the code may be unreadable. Now, let's explain line by line the above code. It does the following tasks:

1. creating two empty lists `it` and `force` with:

   ```
   it     = []
   force  = []
   ```

2. opening the `data.txt` file with:

   ```
   with open('data.txt', 'r') as f:
   ```

The above line is not easy to understand. First, we use the `with` Python keyword for creating a new code block. The `with` statement is used here to create a new *context*: the nested block is executed since the file is correctly opened and read. The file is automatically closed after the `with` statement. Usage of `with` statement is highly recommended for reading or writing files.
Then, the built-in function `open()` is used for opening the related file. The first argument `'data.txt'` is the path to the file and the second argument is the opening mode. Here, the `'r'` string tells to open the file in *read-only* mode.
Finally, the `f` alias is chosen for the file.

3. reading line by line the file in a `for` statement:

```
    for line in f:
```

The `line` variable is a string that contains the related line of the data file.

4. ignoring lines where the character "#" is present:

```
    if not '#' in line:
```

5. splitting the related line thanks to the `split()` method of the string class:

```
        data = line.split()
```

The `data` variable is a list the contains four strings. As example, the first data line gives:

```
['0', '0', '-1.2261431322e+01', '-4.6449842528e-01']
```

6. Storing the first and third elements of `data` in the `it` and `force` lists thanks to the `append()` list method:

```
        it.append(int(data[0]))
        force.append(float(data[2]))
```

You can note the usage of the `float()` and `int()` constructors to force string-to-number conversions.

The above code snippet is a naive implementation... but it works! A more professional program have to manage automatic reading of label, input/output errors or string-to-number conversion errors and so on. But... It works and does the job!

3.2 Writing text files

Writing text file are quit similar to reading file. Suppose that you want to create the following file that contains successive values of trigonometric functions *cos* and *sin*.

```
x     cos(x)                sin(x)
0.0    1.0                  0.0
0.1    0.9950041652780258   0.09983341664682815
0.2    0.9800665778412416   0.19866933079506122
0.3    0.955336489125606    0.29552020666133955
0.4    0.9210609940028851   0.3894183423086505
0.5    0.8775825618903728   0.479425538604203
0.6    0.8253356149096783   0.5646424733950354
0.7    0.7648421872844885   0.644217687237691
0.8    0.6967067093471654   0.7173560908995228
0.9    0.6216099682706644   0.7833269096274834
```

You can write this kind of files with Python thanks to the following code:

```
import math as m

with open('data-output.txt', 'w') as f:
    f.write("x \t cos(x) \t sin(x)\n")
    for i in range(10):
        x = i/10.
        l = "{} \t {} \t {}\n".format(x, m.cos(x), m.sin(x))
        f.write(l)
```

> *Remember that "\n" and "\t" are respectively an end-of-line and a tabulation.*

As for reading file, the `open` built-in function associated with the `with` statement is used here. In addition, you can note the usage of the `'w'` string to specify an opening in *write-only* mode.

The line, just after the `with` statement, writes the header of the text file thanks to the `write()` method of the `f` object. Then, a `for` loop is used to write successively the x, cos(x) and sin(x) values separated by tabulations. You can note the usage of the `format()` string attribute that inserts variable values at a given place (defined by "`{}`") inside a string chain.

Finally, the file is automatically closed after the nested block.

3.3 The numpy module

A fundamental tool for data analysis is numerical arrays. Python gives us the `list` built-in type, but a `list` is an heterogeneous container. It is not able to manage mathematical operations. The `numpy` module provides the `array` class specifically designed for numerical computations.

The `array` class is the fundamental feature of numpy. It mixes the advantage of Python `list` with the performance of C-like-arrays. To guarantee performances, numpy arrays are static. It means that the length (or dimensions) of a numpy array must be given at the construction of the array. You are not able to change its length or dimensions. For instance, numpy arrays do not implement the `append()` method of the `list` class.

Let's see what numpy arrays look like. You can build a numpy array from a Python list as:

```
>>> import numpy as np
>>> np.array([1,2,3,4,5,6,7,8,9,10])
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Multiple dimensional arrays can be constructed as:

```
>>> np.array([ [1,2,3,4,5] , [6,7,8,9,10] ])
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
>>> # an other example
>>> np.array([ [1,2], [3,4], [5,6], [7,8], [9,10] ])
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

You can apply mathematical functions to arrays thanks to numpy functions. These functions are named `ufunc`. For instance, the code below computes the square of each value contained in the array:

```
>>> array = np.array([1,2,3,4,5,6,7,8,9,10])
>>> np.power(array,2)
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

The bracket operator `[]` is available with numpy arrays. For instance:

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

You can iterate over a numpy array such as Python list:

```
>>> array = np.array([1,2,3,4])
>>> for val in array:
...     print(val)
...
1
2
3
4
```

To keep good performances, you must always prefer, if it is possible, to apply numpy functions to a whole numpy array rather applying an operation on each value of an array.

Numpy provides the `arange()` function similar to the built-in `range()` function. It allows to build quickly number suites as:

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

With `arange()`, you can use floating numbers as:

```
>>> np.arange(0, 1, 0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

If you prefer to specify directly the length of the arrays you can use the `linspace()` function rather than `arange()`:

```
>>> np.linspace(0, 1, 11)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

You can directly construct and fill an array with zero or unit values thanks to the `zeros()` or `ones()` functions. You must pass the dimension of the arrays as:

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> np.ones(10)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> np.zeros( (4,4) )
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
>>> np.ones( (4,4) )
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

By default, the `zeros()` or `ones()` functions construct arrays of float. You can check the type contained by numpy arrays thanks to the `dtype` attribute:

```
>>> arr = np.ones((4,4))
>>> arr.dtype
dtype('float64')
```

> ℹ *Note that numpy arrays are homogeneous. It means that all the items contained by the array have the same type. That is the price of performance!*

You can specify the type at the construction with the `dtype` optional argument as it follows:

```
>>> np.ones((10), dtype='int32')
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int16)
>>> np.arange(10, dtype='float64')
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

You can use also random numbers for building array as:

```
>>> np.random.random( (2,2) )
array([[0.36510437, 0.39397511],
       [0.59814128, 0.99681571]])
```

The following code highlights useful trick to get some information from an array:

```
>>> a = np.random.random( (2,2,2) )
>>> a.ndim # number of dimension
3
>>> a.shape # length of each dimension
(2, 2, 2)
>>> a.size # total number of item
8
>>> a.dtype.name # item type
'float64'
>>> a.itemsize # size in byte of item: float64 = 8 bytes
8
```

You can reshape a numpy array with the `reshape()` method as:

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

You can compute the min, max, standard deviation and mean values of an array as it follows:

```
>>> a = np.arange(16).reshape((4,4))
>>> print(a)
[[ 0  1  2  3]
```

```
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> a.min() # min value of array
0
>>> a.max() # max value of array
15
>>> a.std() # standard deviation of array
4.6097722286464435
>>> a.mean() # average value of array
7.5
>>> a.sum() # sum all items
120
```

For multiple dimensional arrays, you can specify the axis to perform these operations. For instance:

```
>>> a = np.arange(16).reshape((4,4))
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> a.min(axis=0) # returns the min of each column
array([0, 1, 2, 3])
>>> a.min(axis=1) # returns the min of each line
array([ 0,  4,  8, 12])
```

To be more comfortable, we can use the `reshape()` numpy method for the last operation:

```
>>> a.min(axis=1).reshape(4,1)
array([[ 0],
       [ 4],
       [ 8],
       [12]])
```

Numerical operations such as multiplications, additions (and so on), between arrays are available. The following example shows an element wise addition between two arrays:

```
>>> a = np.arange(0, 10)
>>> print(a)
[0 1 2 3 4 5 6 7 8 9]
>>> b = np.arange(10, 20)
>>> print(b)
[10 11 12 13 14 15 16 17 18 19]
>>> a + b
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

More advanced operations are available such as *dot products*:

```
>>> a = np.arange(0, 10)
>>> b = np.arange(10, 20)
>>> np.dot(a,b)
735
```

The matrix concept that came from linear algebra is also implemented in numpy. Several ways can be used to construct matrices. For instance:

```
>>> np.matrix([ [1,2], [3,4] ]) # construction with python list
```

```
matrix([[1, 2],
        [3, 4]])
>>> np.matrix('1 2; 3 4') # matlab like construction
matrix([[1, 2],
        [3, 4]])
```

You can transpose or get the inverse of a matrix with the `T` and `I` attributes as:

```
>>> a = np.matrix('1 2; 3 4')
>>> a.T
matrix([[1, 3],
        [2, 4]])
>>> a.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

A very common demand for scientific computation is to solve linear systems. Suppose the following linear system:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \text{ with } \mathbf{A} = \begin{pmatrix} 2 & 3 & -1 \\ 1 & -1 & 1 \\ 2 & -3 & 1 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} -1 \\ 4 \\ 3 \end{pmatrix}.$$

You should want to compute the unknown vector $\mathbf{x}$. You can solve this problem as:

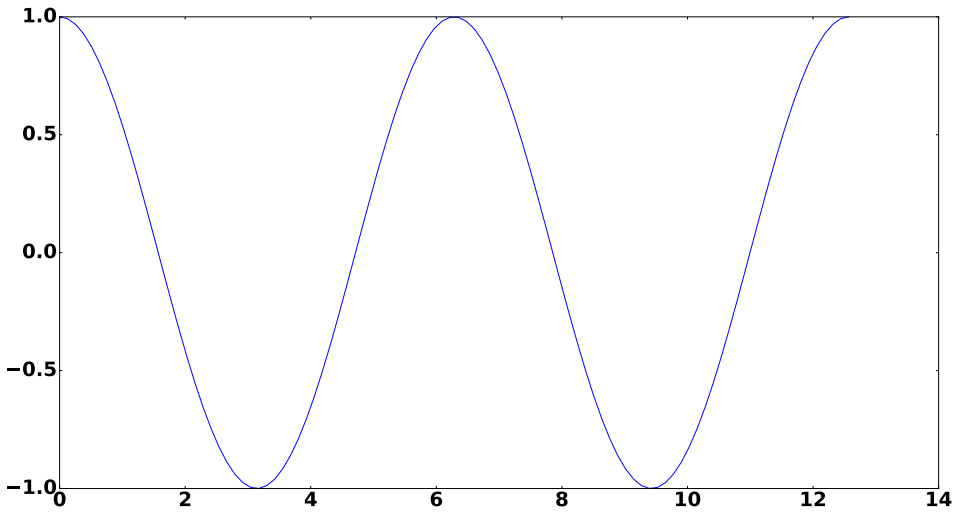
```
>>> A = np.matrix('2 3 -1; 1 -1 1; 2 -3 1')
>>> b = np.matrix('-1; 4; 3')
>>> A.I*b # a first way to compute x
matrix([[0.5 ],
        [0.75],
        [4.25]])
>>> np.linalg.solve(A,b) # a second way to compute x
matrix([[0.5 ],
        [0.75],
        [4.25]])
```

A common reproach is the lack of performances of Python. Here, the problem is solved because you should consider numpy as a *binding* to the well known *Blas* and *Lapack* Fortran libraries. These libraries were extensively optimized by scientists since more than thirty years. In fact, the numpy computations are performed by these Fortran libraries and not by native Python libraries. This design is largely used with Python. It combines the advantage of both languages: Python fair-using with the performances of libraries programmed with fast languages such as Fortran, C or C++.

Note that numpy is not only dedicated to linear algebra and arrays. This module implements numerical treatments and methods such as signal processing, interpolation, stats, polynomial and so on... It is not possible to detail here the whole features, the official documentation is about 1,500 pages!

3.4 Plotting with matplotlib

The `matplotlib` module is the right tool for drawing 2D or simple 3D graphs. Several kind of charts are implemented: point cloud, histogram and so on... To get a quick overview, you can take a look at the matplotlib gallery `https://matplotlib.org/gallery.html`. If you combine matplotlib with numpy you are able to make powerful data analyses with smart charts. For example, if you want to draw the following chart that plots $f = sin(x)$ from 0 to 4π,



the related code is:

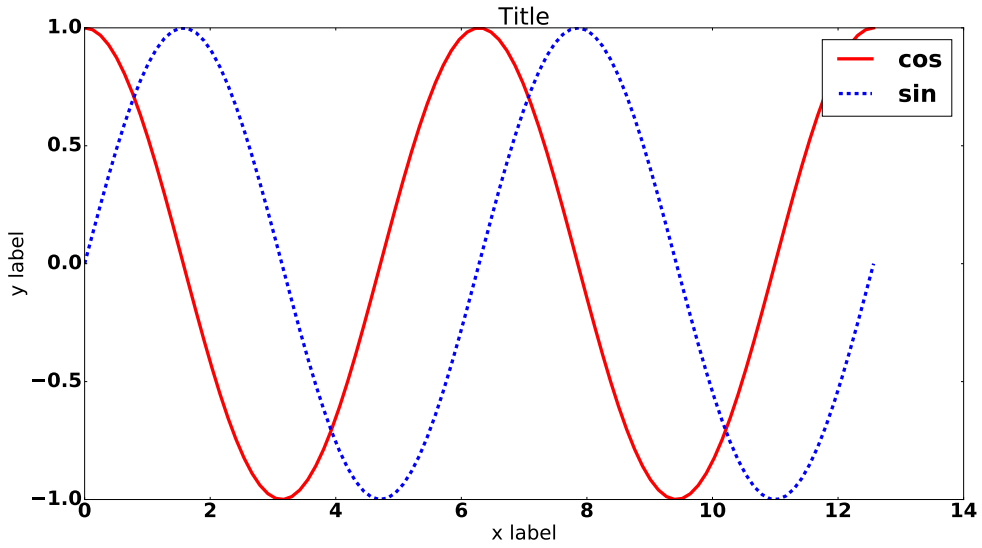
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 100.)
y = np.cos(x)
plt.plot(x,y)
plt.show()
```

The first line `import matplotlib.pyplot as plt` imports the pyplot module from the matplotlib package. Note that the `matplotlib.pylab` module is also available but it is less and less used. Then, numpy arrays are used to build the `x` and `y` data arrays. The line `plt.plot(x,y)` tells to matplotlib to prepare the chart and, finally, the plot is displayed with `plt.show()`. Note that matplotlib displays interactive charts. You are able to zoom, save image in several formats, etc...

You can easily add label, title or legends. For example, the following code add another line plot with some labels:

```
import matplotlib.pylab as plt
import numpy as np

x = np.linspace(0, 4*np.pi, 100.)
plt.plot(x, np.cos(x), 'r-' , label='cos', lw=5)
plt.plot(x, np.sin(x), 'b--', label='sin', lw=5)

plt.legend()
plt.title('Title')
plt.xlabel('x label')
plt.ylabel('y label')
plt.show()
```

The above code gives the following chart:



It is not possible to present here the whole matplotlib features. A large number of options exists! My advice is: if you have a precise idea of what kind of chart you want, you can browse the official gallery and choose a chart closest to your need. After that, you are free to copy/paste the related Python code and edit it for adapting the code to your inquiry.

3.5 Advanced treatments with scipy

As a first approach, the scipy module can be viewed as an extension of numpy. In fact, some treatments such as Fast Fourier Transform are available in both numpy and scipy. You should consider numpy as a very stable package and the scipy package as an advanced package but less stable than numpy. Scipy is made of the following modules:

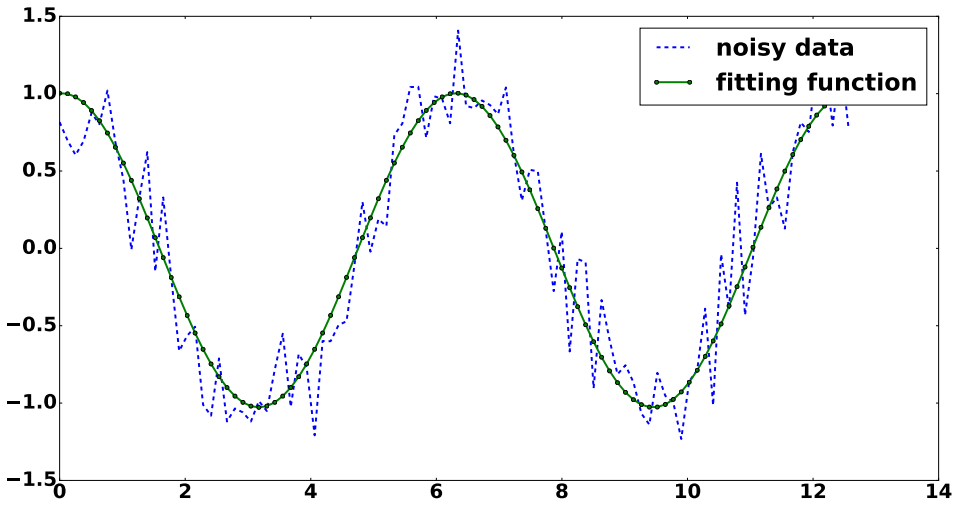
- Special functions in `scipy.special`
- Integration in `scipy.integrate`
- Optimization in `scipy.optimize`
- Interpolation in `scipy.interpolate`
- Fourier Transforms in `scipy.fftpack`
- Signal Processing in `scipy.signal`
- Linear Algebra in `scipy.linalg`
- Sparse Eigenvalue Problems with ARPACK
- Compressed Sparse Graph Routines in `scipy.sparse.csgraph`
- Spatial data structures and algorithms in `scipy.spatial`
- Statistics in `scipy.stats`
- Multidimensional image processing in `scipy.ndimage`
- File input/output in `scipy.io`

As for numpy and matplotlib, it is not possible to describe here the whole scipy package. The following code highlights one of the large number of scipy features: the non linear least square method that comes from the `optimization` package. The non linear least square method is used here for fitting a noisy set of points by a trigonometric function.

```
import numpy as np
import matplotlib.pyplot as plt

# generates noisy data
```

```
x = np.linspace(0, 4*np.pi, 100.)
y = np.cos(x) + np.random.randn(len(x))/4

# use non linear square method with 'curve_fit'
from scipy.optimize import curve_fit

# define here the fitting function
def f(x, a0, a1, a2, a3):
    return a0 + a1 * np.cos(a2 + a3*x)

# use curve_fit to get the 'best' values of a0, a1, a2 and a3
popt, pcov = curve_fit(f, x, y)
a0, a1, a2, a3 = popt

# plot results
z = f(x, a0, a1, a2, a3)
plt.plot(x, y, '--', lw=3, label='noisy data')
plt.plot(x, z, 'o-', lw=3, label='fitting function')
plt.legend()
plt.show()
```

The above code gives the following output



4 Advanced libraries and concepts

4.1 A short introduction to *Pandas*, a library for big data analysis

Pandas is a Python library dedicated to data analysis. I am not a Panda specialist but, in my mind, Pandas is a good tool when the amount of data becomes high especially if you have statistical data that can be written in a table in the `[column]x[line]` format. It allows us fast and easy filtering and sorting of your data. Let's start with a very practical example. Imagine that you want to buy a given house in France and you want to check if the price is correct... or not!

To do that, you can download the official french government data that record the house selling for the past four years. These data embed all the commercial transaction from the last four years including information about transaction prices, locations of house, their surfaces and so on... For

example, you want to buy a 80 m^2 flat in Limoges ;) and the price is 200,000€. Let's check if the price is good.

To do such thing, let's go on the official website https://cadastre.data.gouv.fr/dvf, for downloading the last data and unzip it[1] Finally, you must obtain a text file named `"valeursfoncieres-YEAR.txt"`.

If you look at the header of this text file, you will see something like:

```
Code service CH|Reference document|1 Articles CGI|2 Articles CGI|3 Articles CGI|4 Articles CGI|5
    Articles CGI|No disposition|Date mutation|Nature mutation|Valeur fonciere|No voie|B/T/Q|Type de
     voie|Code voie|Voie|Code postal|Commune|Code departement|Code commune|Prefixe de section|
    Section|No plan|No Volume|1er lot|Surface Carrez du 1er lot|2eme lot|Surface Carrez du 2eme lot
    |3eme lot|Surface Carrez du 3eme lot|4eme lot|Surface Carrez du 4eme lot|5eme lot|Surface
    Carrez du 5eme lot|Nombre de lots|Code type local|Type local|Identifiant local|Surface reelle
    bati|Nombre pieces principales|Nature culture|Nature culture speciale|Surface terrain
|||||||000001|08/01/2016|Vente|40000,00|77||RUE|0560|TONY REVILLON|1750|SAINT-LAURENT-SUR-SAONE
    |01|370||A|253||4|41,55|||||||||1|2|Appartement||50|2|||
|||||||000001|11/01/2016|Vente|1677,00||||B011|LES BROTTEAUX|1160|VARAMBON|01|430||C
    |1043|||||||||||0||||||L||1486
|||||||000001|11/01/2016|Vente|1677,00||||B011|LES BROTTEAUX|1160|VARAMBON|01|430||C
    |1157|||||||||||0||||||L||3904
|||||||000001|11/01/2016|Vente|1677,00||||B011|LES BROTTEAUX|1160|VARAMBON|01|430||C
    |1159|||||||||||0||||||L||1779
```

In fact, the data are presented in a special CSV (Comma Separated Values) format. The first (long) line contains the label of each column separated by the `'|'` character instead of the more standard ones: comma `','` or semi-column `';'` characters. A second comment concerns the usage of the comma as decimal operator instead of the widely used dot `'.'` character.

Now, we can use the `read_csv()` function that comes from the Pandas module to read this csv file.

```
>>> import pandas as pd
>>> df = pd.read_csv('valeursfoncieres-2018.txt', sep='|', decimal=',')
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

As you can see, the `read_csv()` function returns a `DataFrame` object. The `DataFrame` class is the main data container of Pandas. A huge number of methods and attributes are available for the `DataFrame` class. It is not possible to make an exhaustive description here. A common request is to get a list of the available columns with:

```
>>> df.columns
Index(['Code service CH', 'Reference document', '1 Articles CGI',
       '2 Articles CGI', '3 Articles CGI', '4 Articles CGI', '5 Articles CGI',
       'No disposition', 'Date mutation', 'Nature mutation', 'Valeur fonciere',
       'No voie', 'B/T/Q', 'Type de voie', 'Code voie', 'Voie', 'Code postal',
       'Commune', 'Code departement', 'Code commune', 'Prefixe de section',
       'Section', 'No plan', 'No Volume', '1er lot',
       'Surface Carrez du 1er lot', '2eme lot', 'Surface Carrez du 2eme lot',
       '3eme lot', 'Surface Carrez du 3eme lot', '4eme lot',
       'Surface Carrez du 4eme lot', '5eme lot', 'Surface Carrez du 5eme lot',
       'Nombre de lots', 'Code type local', 'Type local', 'Identifiant local',
       'Surface reelle bati', 'Nombre pieces principales', 'Nature culture',
       'Nature culture speciale', 'Surface terrain'],
      dtype='object')
```

If you want to see a preview of the first values of the `DataFrame`, you can use the `head()` method as follows:

[1] In fact, this is not a `zip` file this a `tar.gz` file. For uncompressing it, you can use 7-zip on Windows.

```
>>> df.head()
   Code service CH  Reference document  ...  Surface terrain
0              NaN                 NaN  ...              NaN
1              NaN                 NaN  ...              NaN
2              NaN                 NaN  ...            949.0
3              NaN                 NaN  ...            420.0
4              NaN                 NaN  ...            949.0

[5 rows x 43 columns]
```

To get the whole data list of a column, you can use the bracket operator `['name']` as:

```
>>> df['Surface terrain']
0               NaN
1               NaN
2             949.0
              ...
2338999         NaN
2339000         NaN
2339001         NaN
Name: Surface terrain, Length: 2339002, dtype: float64
```

Here, you can note that:

- at the end of the output, some usefull information are displayed: type, length, etc. of the given column.
- a lot of `NaN` (Not a Number) exists. This is probably caused by the lack of information.You must take care about this point later during data processing.

Now, we will use the `'Code postal'` (zip code) column to filter data for the Limoges city. The zip code of Limoges is 87000. Let's try something:

```
>>> df['Code postal'] == 87000
0          False
1          False
2          False
           ...
2338999    False
2339000    False
2339001    False
Name: Code postal, Length: 2339002, dtype: bool
```

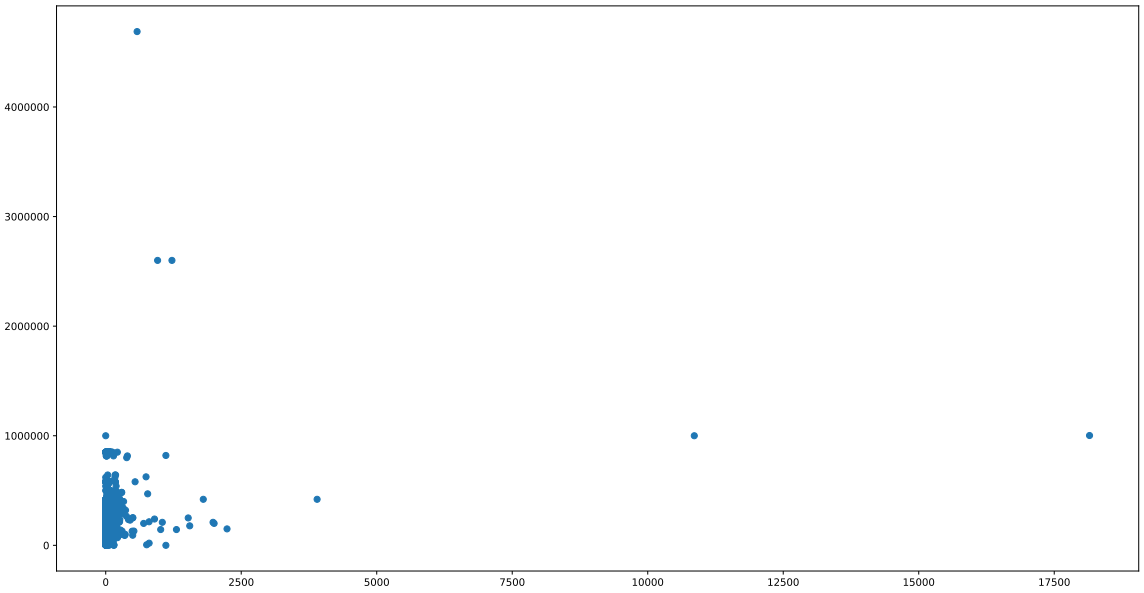
Here, the condition `df['Code postal'] == 87000` returns a list of Boolean. We can use this result to index our records as:

```
>>> df_lim = df[df['Code postal'] == 87000]
>>> type(df_lim)
<class 'pandas.core.frame.DataFrame'>
>>> df_lim['Code postal']
1992604    87000.0
1992605    87000.0
1992635    87000.0
            ...
2014120    87000.0
2014121    87000.0
2014130    87000.0
Name: Code postal, Length: 3453, dtype: float64
>>>
```

Here, we apply a filter. It returns a `DataFrame` named `df_lim` that contains only the record of the Limoges city. Now, we play with the data. Let's plot the price versus the surface.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(df_lim['Surface reelle bati'], df_lim['Valeur fonciere'], 'o')
>>> plt.show()
```

It gives the following plot:



As you can see, the given plot is not so smart. We need to apply some filters able to remove crazy points. We will apply two filters:

1. surfaces of the houses must be between 20m^2 and 400m^2.

```
>>> df_lim = df_lim[df_lim['Surface reelle bati'] >= 20]
>>> df_lim = df_lim[df_lim['Surface reelle bati'] <= 400]
```

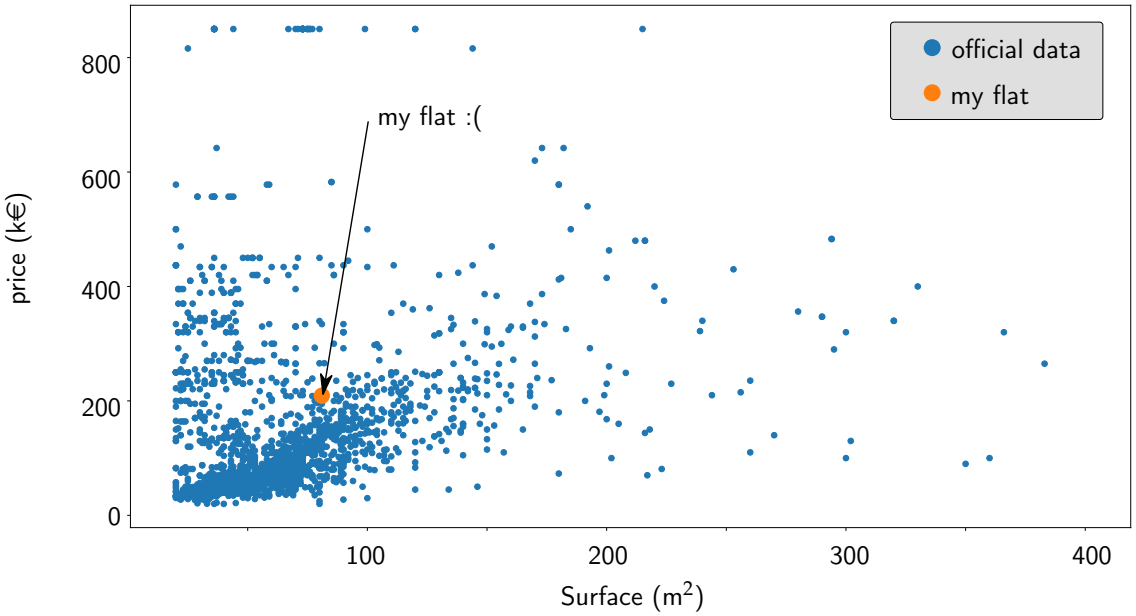
2. prices of the houses must be between 20k€ and 1,000k€.

```
>>> df_lim = df_lim[df_lim['Valeur fonciere'] >= 20e3]
>>> df_lim = df_lim[df_lim['Valeur fonciere'] <= 1000e3]
```

Now, we can replot the given data frame with:

```
>>> plt.plot(df_lim['Surface reelle bati'], df_lim['Valeur fonciere']*1e-3, 'o', label='offical')
>>> plt.plot([80], [200], 'o', label='my flat')
>>> plt.xlabel('surface (m2)')
>>> plt.ylabel(u'price (kilo euro)')
>>> plt.legend()
>>> plt.show()
```

If we add smart drawing to the given plot, it gives the following chart.



As you can see on the above graph, the 80 m^2 flat in Limoges sells for 200,000€ is in the highest range of the obtained point cloud. It means that your flat is quite expansive if you consider all the transactions of the Limoges city in the last year. Based on this observation, you are able to negotiate the price of the flat with the owner! If the owner is really hard, you can make more sophisticated statistical data processing such as linear regression, bar plotting, and so on... If the owner is a scientist, you may convince him!

To conclude, as you can see in this section, the Pandas library is really easy and helpfull for data processing. On my laptop, the duration of the file opening is only 6.5s for a 298 Mo data file. As comparison, try to open it with the Excel software, you should be surprised!

4.2 A short introduction to *Sympy* for symbolic calculation

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.

As an example, imagine that you want to study the following function:

$$f(x) = \ln(x+1) - 2$$

A preliminary step is to import the whole sympy module in the current Python environment:

```
>>> from sympy import *
```

A second preliminary step is to initialize printing outputs. It renders equations with nice and smart outputs:

```
>>> init_printing()
```

Now, you can use sympy. A first thing is to let sympy recognize x as a mathematical symbol:

```
>>> x = symbols('x')
```

> *Note that you can export several symbols in one time as:*
>
> ```
> >>> x, y = symbols('x y')
> ```

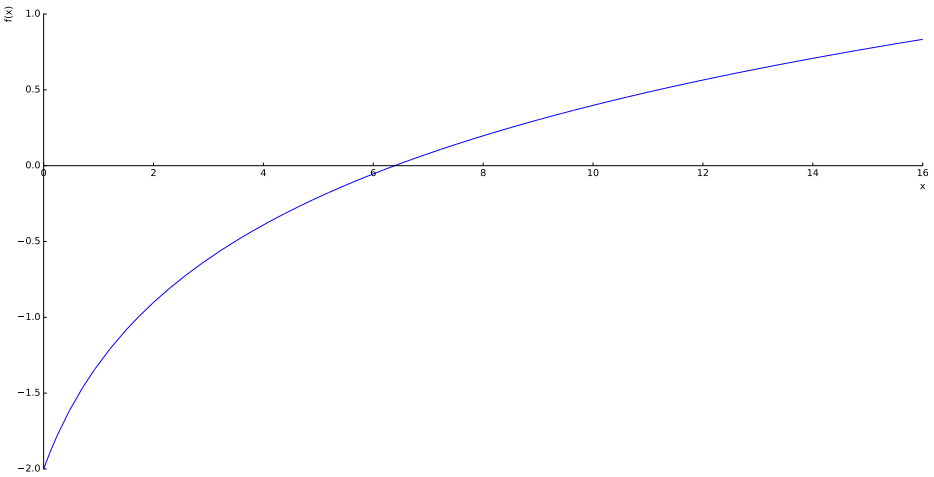
You can simply define the f function as ;

```
>>> f = ln(x+1)-2
>>> f
log(x + 1) - 2
```

For plotting this function in the [0,16] range, you can type:

```
>>> plot(f, (x, 0, 16))
```

It gives the following chart:



Suppose that you want to solve $f(x) = 0$. It is given by:

```
>>> x0 = solve(f,x)
>>> x0
[      2]
[-1 + e ]
```

Note that the `solve()` returns a list of solutions. Here, only one solution is available. We can overwrite `x0` Python variable as:

```
>>> x0 = x0[0]
>>> x0
      2
-1 + e
```

So, the solution of $f(x) = 0$ is

$$x_0 = -1 + e^2$$

Now, suppose that you want to compute the first derivative $g(x)$ and the first integral $h(x)$ of $f(x)$. It is given by:

```
>>> g = diff(f,x)
>>> g
  1
-----
x + 1
>>> h = integrate(f,x)
>>> h
x.log(x + 1) - 3.x + log(x + 1)
```

So, the above code gives:

$$f'(x) = g(x) = \frac{1}{x+1}$$
$$\int f(x) = h(x) = \ln(x+1) - 3x + \ln(x+1)$$

Now, suppose that you want to compute $h(x_0)$. It is given thanks to the `subs()` function as:

```
>>> h.subs(x,x0)
   2
- e  + 3
```

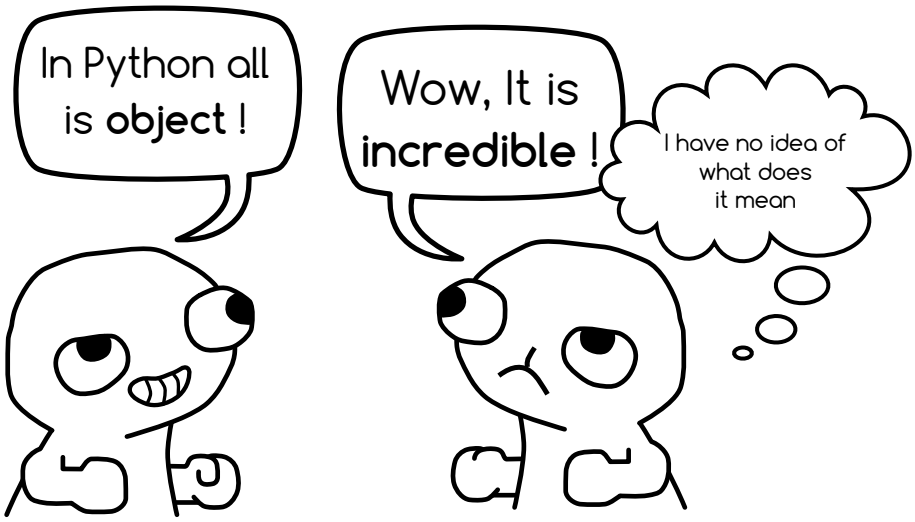
Finally, if you want a floating-point approximation of this expression, you can use the `evalf()` function as:

```
>>> h.subs(x,x0).evalf()
-4.38905609893065
```

This quick introduction gives you an overview of the sympy module. You can use sympy as any Python modules inside a Python script. You are free to combine all these scientific modules to build powerful data treatments and analyses! Now, you must learn by yourself and improve your skills in Python. To help you in training yourself, you can study the practical exercises available at the end of this document.

4.3 Object oriented programming

If you already know Python you probably have already heard "In Python all is object !" and your feeling was probably:



This section will introduce the *Object Oriented Programming* (OOP) concept. OOP is one of the coding paradigm or, if you prefer, one of the coding "*style*". You don't have to know these paradigms to program something. Many coders mix intuitively these paradigms, depending on the context, without knowing them theoretically.

To be honest, I have some difficulties to explain theoretically what OOP is. For me, a good starting point is to take a simple example to illustrate the OOP concepts. I will introduce OOP through the pacman game (see figure 3). You probably know that this game involves two main types of character: pacman and ghosts.

Figure 3: The pacman game

Imagine that you want to code from scratch this game. Your first task is to code the ghosts. To do correctly this job, you have to enumerate the characteristics of a ghost. For example, a ghost:

- owns a color
- owns a (x,y) position
- can eat pacman
- can move vertically and horizontally

In fact, when you enumerate these characteristics, you are building the ghost *class*. A *class*, that comes from OOP, is a programming entity that enumerates the characteristic of *something*. Here, *something* is *ghost*.

- " Great, but what is an *object*? "
An object is a class *instance*.

- " Good! but what is a *class instance*? "
A class instance (or an object) is one of the ghost in the game. All the ghosts in the game where defined by the same class! The ghost class defines the concept of ghost and a ghost object is one of this ghost.

- " It is a little bit confused, can you give me an example? "
Right, imagine that we want to build the ghost class with Python, it gives:

```
class ghost:
    def __init__(self):
        self.color = color
        self.pos_x = 0
        self.pos_y = 0
```
pacman.py

Now, you can build ghost objects with:

```
>>> import pacman as pc
>>> pc.ghost()
<pacman.ghost object at 0x7f42a694d5c0>
```

In fact, when you type `pc.ghost()` function, you invoke the `__init__()` function of the ghost class. This special function is called at the construction of the object.

> ℹ *Special python functions are generally prefixed and/or suffixed by underscore "_".*

To convince you, we will add some outputs to the `__init__()` function. The ghost class becomes:

```
class ghost:
    def __init__(self):
        self.color = 'red'
        self.pos_x = 0
        self.pos_y = 0
        print("building a new", self.color, "ghost:")
        print(" - my memory address is", hex(id(self)))
        print(" - my type is", type(self))
```
pacman.py

Now, we instantiate a new ghost:

```
>>> import pacman as pc
>>> pc.ghost()
building a new red ghost:
 - my memory address is 0x7f6f1c7dd5c0
 - my type is <class 'pacman.ghost'>
<pacman.ghost object at 0x7f6f1c7dd5c0>
```

As you can see, some outputs are given during object creation. The main trick here is the usage of the `self` Python keyword in class. As you can see, the `self` keyword is simply the address of the created object! So, usage of `self` inside a class is related to the current object instance.

The `self` keyword can be used for adding some *attributes* to class. *Attributes* can be viewed as data attached to objects. The syntax `object.attribute` allows to access to the related attribute. For example, if you want to change the color of a ghost:

```
>>> import pacman as pc
>>> a = pc.ghost()
building a new red ghost:
 - my memory address is 0x7f2c51b3bba8
 - my type is <class 'pacman.ghost'>
>>> a.color
'red'
>>> a.color = 'green'
>>> a.color
'green'
```

Now, suppose that you want to specify the color during the object creation. You can add some parameters to the `__init__` function as it follows:

```
class ghost:
    def __init__(self, col):
        self.color = col
        self.pos_x = 0
        self.pos_y = 0
        print("building a new", self.color, "ghost")
```
pacman.py

It gives:

```
>>> import pacman as pc
>>> pc.ghost('pink')
building a new pink ghost
<pacman.ghost object at 0x7f2c51d57550>
```

As for attributes, you can add functions to classes. In OOP terminology, functions related to classes are called *methods*. For example, you should want to add a method named `move_up()` that increments the ghost's position on y. The ghost class becomes:

```
class ghost:
    def __init__(self, col):
        self.color = col
        self.pos_x = 0
        self.pos_y = 0
        print("building a new", self.color, "ghost")

    def move_up(self):
        self.pos_y += 1
```
pacman.py

Now, we can use the class as it follows:

```
>>> import pacman as pc
>>> g = pc.ghost('pink')
building a new pink ghost
>>> g.pos_y
0
>>> g.move_up()
>>> g.pos_y
1
```

- " What is the interest? We can change directly the `pos_y` attribute from outside the class! "
Yes you right, but you loose semantic. Look at these two codes:

```
>>> import pacman as pc
>>> pink_ghost = pc.ghost('pink')
>>> pink_ghost.move_up()
```

```
>>> import pacman as pc
>>> pink_ghost = pc.ghost('pink')
>>> pink_ghost.pos_y += 1
```

These two codes do exactly the same thing but the first one is more readable. In addition, you can use methods to prevent some non-wanted behaviors. For example, suppose that the maximal y position is 100. Higher values means that the character goes outside the arena and it is forbidden. You can prevent such behavior by adding a condition inside the `move_up()` method as it follows:

```
class ghost:
    def __init__(self, col):
        self.color = col
        self.pos_x = 0
        self.pos_y = 0
        print("building a new", self.color, "ghost")

    def move_up(self):
        if self.pos_y < 100:
            self.pos_y += 1
```
pacman.py

Now, the ghosts are not able to go to a position along y higher than 100.

- " Okay.... but you say at the beginning: *in Python all is object*. What is *all*? "
Yes, in Python all is object. It means that a module is an object and a class itself is an object. Let's see the following example:

```
>>> import pacman as pc
>>> type(pc)
<class 'module'>
>>> type(pc.ghost)
<class 'type'>
>>> type(print)
<class 'builtin_function_or_method'>
```

In this example, you can see that:

- the `pc` module is an object of the `'module'` class,
- the `pc.ghost` class itself is an object of the `'type'` class and
- the `print` builtin function is an object of the `'builtin_function_or_method'` class.

Python is really amazing!

A more advanced feature with OOP is the class *inheritance*. *Inheritance* allows to share in super classes common attributes and methods. These attributes and methods can be shared between classes that inherit from the same super class.

Following our pacman example, suppose that we want to build the `pacman` class. You can see easily that pacman and ghosts share some behaviors: an (x,y) position, displacement capacities and color. So it is interesting to factorize these behaviors in a common super class `character` as it follows:

```
class character:
    def __init__(self, color):
        self.color = color
        self.pos_x = 0.
        self.pos_y = 0.

    def move_up(self):
        if self.pos_y < 100:
            self.pos_y += 1

class ghost(character):
    def __init__(self, color):
        character.__init__(self, color)

class pacman(character):
    def __init__(self):
        character.__init__(self, "yellow")
```
pacman.py

In this example, both `ghost` and `pacman` classes inherit from the super class `character` thanks to the syntax `class subclass(superclass)`.

So, you are able to invoke the `move_up()` method from either `ghost` and `pacman` objects as it follows:

```
>>> import pacman as pc
>>> pacman = pc.pacman()
>>> ghost1 = pc.ghost('pink')
>>> pacman.move_up()
>>> ghost1.move_up()
```

OOP is a powerful tool. Note that Python allows also multiple inheritance. If you want to become a pro OO programmer, you can take a look to the Unified Modeling Language (UML). UML is a powerful tool for visualizing complex object oriented design with standard diagrams. Note that the object oriented approach is implemented in several languages such as C++ or Java.

5 Practical works

Now, you have basic knowledge of Python programming techniques. You must train yourself to improve your skills.

5.1 Guess the number game

Goal	Implement the guess the number game
Prerequisite	language basics
Duration	15 min
Correction	`guess-the-number.py`

This exercise consists in programmings the *guess-the-number* game.

- *task1.* randomly choose an integer number in [0,100] range. You can use the `random` module.
- *task2.* ask users to give a number in this range. If input is out-of-range, you must specify it.
- *task3.* program must tell to users if the entry is lower or higher than the random hidden number.
- *task4.* program stops if users guess the right number.

The output must look like:

```
give a number: 50
your entry is lower than hidden number
give a number: 75
your entry is higher than hidden number
give a number: 60
your entry is higher than hidden number
give a number: 55
your entry is lower than hidden number
give a number: 57
your entry is lower than hidden number
give a number: 58
your entry is lower than hidden number
give a number: 59
You guess the number. The mystery number was '59'
```

You can improve your program by using a maximal number of tentative.

5.2 The hangman game

Goal	Implement the hangman game
Prerequisite	language basics
Duration	30 min
Correction	`hangman.py`

The goal of this game is to guess a mystery word.

- *task1.* randomly choose a mystery word in a given word list.
- *task2.* ask users to give a character.
- *task3.* if the given character is in the word, the related character must be revealed.
- *task4.* the game ends if the word is completely revealed.

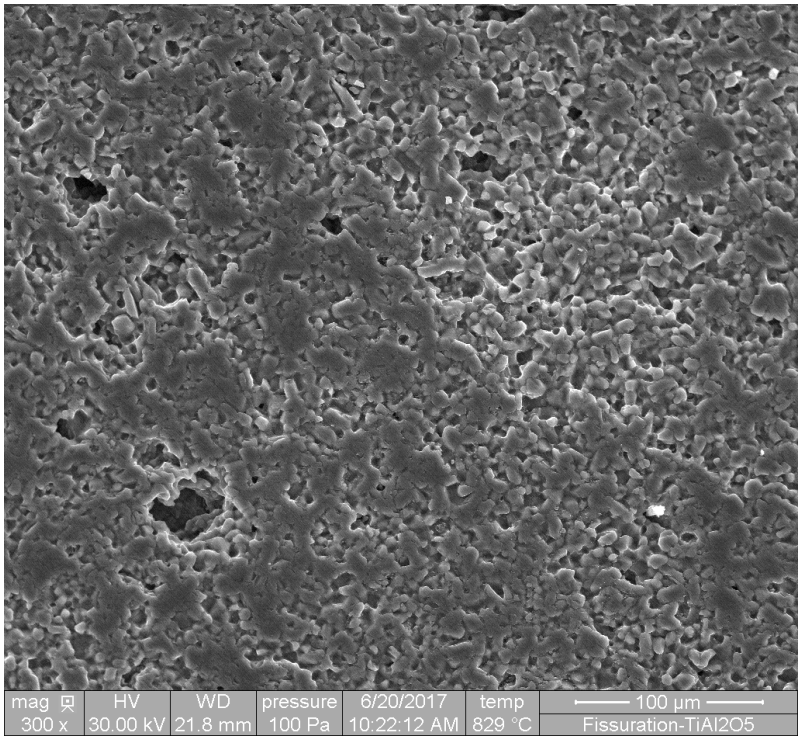
The outputs of your program must looks like:

```
current word is '['_', '_', '_', '_', '_']'
give a character: l

current word is '['_', '_', 'l', 'l', '_']'
give a character: h

current word is '['h', '_', 'l', 'l', '_']'
give a character: e

current word is '['h', 'e', 'l', 'l', '_']'
give a character: o

YOU WIN !!
The mystery word was 'hello'
```

5.3 Hacking image files

Goal	Extract hidden information from image files
Prerequisite	language basics and reading/writing files
Duration	1 hour
Correction	`read-img/read-img.py`

The objective of this training exercise is to extract some interesting meta-data from image files. Note that this exercise is inspired from a real case! First, you have to download the following archive `https://gitlab.com/damien.andre/learning-python-for-science/blob/master/script/read-img/img.zip` that contains several image files. These images were given by Scanning Electron Microscope on a Al2TiO5 material. These images look like:



If you take a look at these images, you can see at the bottom some useful information: magnitude, pressure, temperature and so on... In fact, these information are hidden inside the files. Open with a **text editor** one of these image files and take a look at the end of the file. What did you observe?

task1. write a script that reads an image file line by line in the ascii format and that displays the result with the `print()` function. To avoid any error while reading files, you can set the optional argument `errors` to the `'ignore'` value of the `open(...)` built-in function as it follows:

```
open('yourfile', 'r', encoding='ascii', errors='ignore') # for python3
```

⚠ *Be aware, with python2, you must do*

```
open('yourfile', 'rb')
```

task2. thanks to a condition, build a filter that displays only the lines that contain the temperature and the pressure in the vacuum chamber.

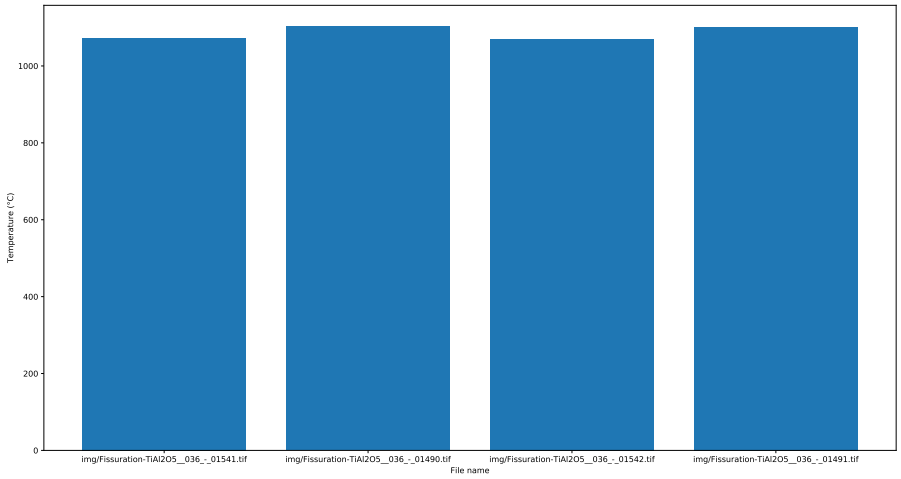
task3. thanks to the `replace()` method of the `str` class, extract the temperature and pressure and convert as floating numbers.

Imagine that you have a huge number of images. in order to build a global image database, you want to extract all these information.

task4. extract, for each image file, the related temperature and pressure. Store these information in a dictionary, lists or tuples . To parse all files in a given directory, you can use the `glob` module as this:

```
import glob
for filename in glob.glob('*.tif'):
   print filename
   # do your stuff here
```

task5. Plot, with 2 bar diagrams, the evolution of temperature and pressure related to image files. These diagrams must look like:



Suppose that we want to build a database file that provides, for each given file, the related temperature and pressure. A naive implementation of this database may look like:

```
Filename        Temperature        Pressure
xxxxx.tif       xxxx               xxxxx
xxxxx.tif       xxxx               xxxxx
...
```
img.db

task6. from the above listing, build a script that automatically write this database. You could name this database "`img.db`".

Now, we would like to use a more advanced file format. A popular format for this kind of little database is the *json* format that looks like:

```
{
  "img/Fissuration-TiAl2O5__036_-_01542.tif": {
    "temperature": 1070.18,
    "pressure": 99.6605
  },
  "img/Fissuration-TiAl2O5__036_-_01491.tif": {
    "temperature": 1101.76,
    "pressure": 100.042
  },
  "img/Fissuration-TiAl2O5__036_-_01541.tif": {
    "temperature": 1072.74,
    "pressure": 100.042
  },
  "img/Fissuration-TiAl2O5__036_-_01490.tif": {
    "temperature": 1102.57,
    "pressure": 100.423
  }
}
```

img.json

task7. with the `json` module, able to dump the content of a python dictionary, build a script that writes this json file automatically.

5.4 Legacy function vs universal function (ufunc)

Goal	Universal function benchmark
Prerequisite	language basics, numpy
Duration	20 min
Correction	`bench-ufunc.py`

The goal of this exercise is to benchmark a legacy function versus a numpy universal function (ufunc).

task1. thanks to the `np.arange` function, build an array `x` that contains 10^7 elements.

task2. implement a legacy function `sin_py()` that:

- takes as argument the `x` array
- returns a numpy array that contains the sinus of each element

task3. execute this function for the `x` array. Monitor the elapsed time thanks to the `time` module.

task4. execute the `np.sin()` ufunction.

Your output should give something like:

```
elapsed time for sin_py(x) is 1.7576186656951904 s
elapsed time for np.sin(x) is 0.17843055725097656 s
```

5.5 Post-treating student rating

Goal	Quick statistical treatment
Prerequisite	language basics, numpy
Duration	20 min
Correction	`grading.py`

This exercise highlights numpy array methods. Here, the data comes from student rating.

> *task1.* thanks to the `np.random.normal`, generate an array of 500 values in the [0,20] range corresponding to student rating.

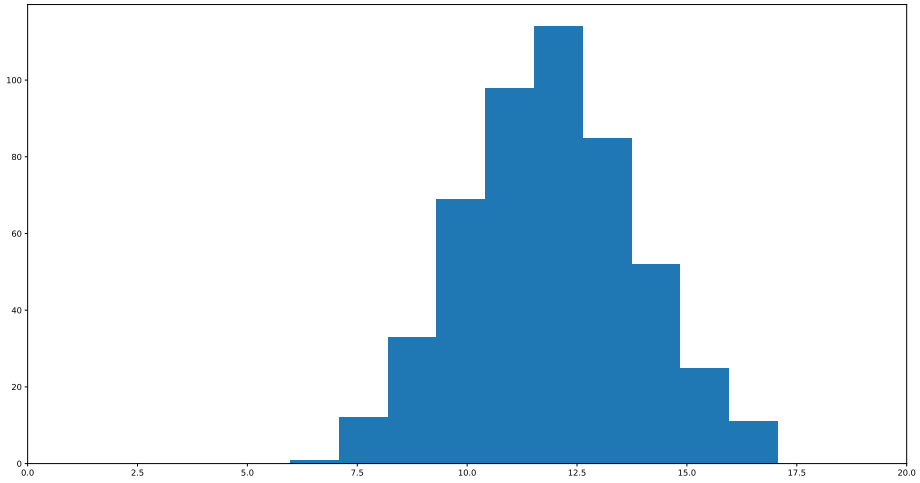
> *task2.* thanks to numpy array methods, compute the following data: student number (must be 500), max value, min value, mean value, standard deviation value, cumulative summation, number of student higher than 15 and sorted values.

Your output should give something like:

```
rating number                  ; 500
min value                      : 6.017527039633518
max value                      : 17.79272432969885
mean value                     : 12.04497827181669
standard dev                   : 1.9866937935737548
cumulative sum                 : 6022.489135908345
number of values higher than 15: 34
sorted rating  [ 6.99550988  7.5029463   7.76531536  ... 18.06266806]
```

> *task3.* plot the related histogram in the [0,20] range with 10 classes.

You must obtain something like this.



5.6 Summation function evaluation

Goal	Compute a summation function
Prerequisite	language basics, numpy, matplotlib
Duration	10 minutes
Correction	`sum.py`

The objective is simple: compute and plot the following function in the $[-\frac{\pi}{2}; \frac{\pi}{2}]$ range

$$f(x) = \sum_{q=1}^{100} \cos(qx)$$

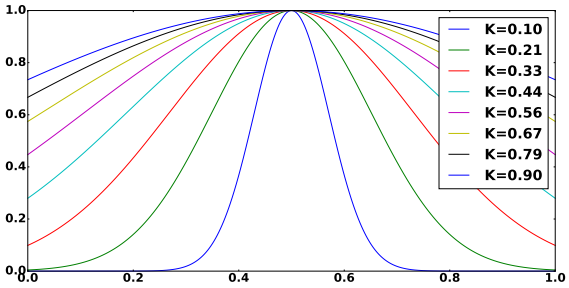
5.7 Simple gaussian function

Goal	Display gaussian function
Prerequisite	language basics, numpy, matplotlib
Duration	10 minutes
Correction	`gauss.py`

The objective is simple: compute and plot the following gaussian function in the $[0;1]$ range for several value of the K factor.

$$f(x) = e^{-\left(\frac{x-0.5}{K}\right)^2}$$

You must obtain something like this:



5.8 2D gaussian function and image

Goal	
Prerequisite	language basics, numpy, matplotlib
Duration	10 minutes
Correction	`gauss-2D.py`

The objective of this training exercise is to create an artificial image from a 2D function. Here, we will use the gaussian function given by:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{1}$$

where σ is the standard deviation and μ the mean of the related statistical distribution.

task1. Implement the `gauss` function . This function should returns a numpy array. The prototype of this functions should be:

```
def gauss(x, mu, sigma):
    ...
```

Now, we will implement a 2D version of the gaussian function:

$$G(x,y) = g_1(x) \times g_2(y) \tag{2}$$

task2. Copy/paste the following code that implements the `gauss_2D` function .

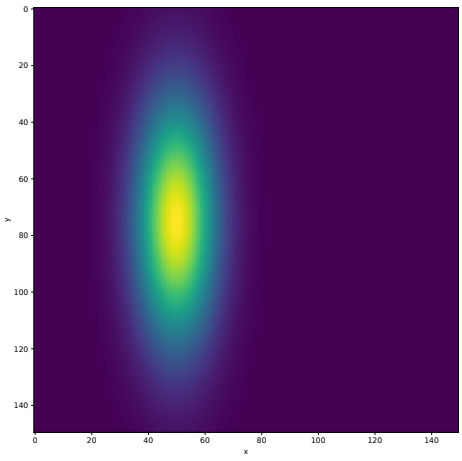
```
def gauss_2D(x, y, mux, muy, sigx, sigy):
    return gauss(x,mux,sigx)*gauss(y,muy,sigy)
```

task3. Thanks to the `numpy.arange` function, build two arrays `x` and `y`

task4. Use the `gauss_2D` function for computing a 2D array from `x` and `y`. Take care about the `x` and `y` shape!

task5. Plot the result with the `plt.imshow` function

You must obtain something like this:



5.9 Linear regression with least square method

Goal	
Prerequisite	language basics, numpy, scipy, matplotlib
Duration	10 minutes
Correction	`lsq.py`

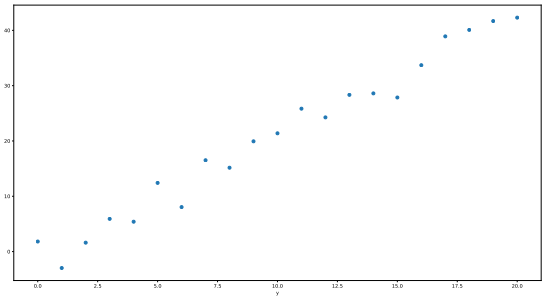
The objective of this training exercise it to implement a least square regression for fitting data with a linear function.

task1. Download the data `lsq.txt` and place it on your current python working directory. Read this data file using the following code snippet:

```
x,y = np.loadtxt("lsq.txt", unpack = True)
```

task2. Plot the `x` array versus the `y` array

You should obtain something like this:



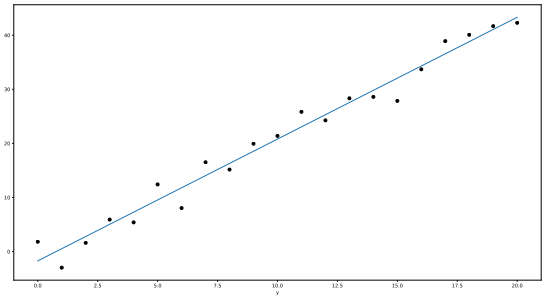
task3. Implement a linear function. The prototype should be:

```
def linear_func(x, a, b):
    ...
```

task4. Use the `optimize.curve_fit` function that comes from the `scipy.optimize` module to get best fit values of the `a` (slope) and `b` (intercept) coefficients of the `linear_func`

task5. Plot on the same graph the best fit curve and the data points.

You must obtain something like this:



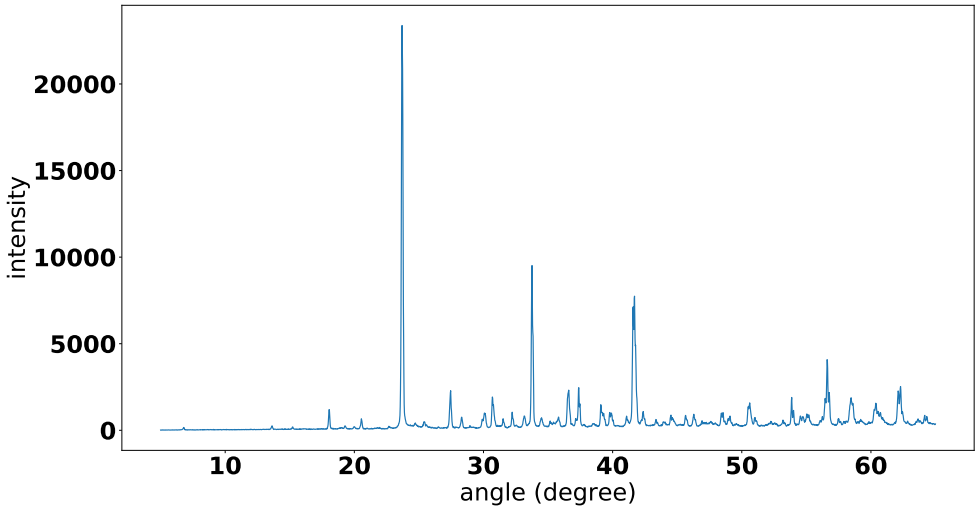
5.10 Post-treating X-ray diffraction raw data

Goal	
Prerequisite	language basics, i/o, numpy, scipy, matplotlib
Duration	1.5 hours
Correction	`drx.py`

The objective of this exercise it to post-treat experimental data that comes from x-ray diffraction. Here the objective is to measure very precisely the shifting of a given peak of different sample.

task1. Download the data `drx-sample1.xy`, `drx-sample2.xy`, `drx-sample3.xy`, `drx-sample4.xy` and `drx-sample5.xy` and place them on your current python working directory. Read these data with your favorite text editor.

task2. Open the first file and push data in `numpy.array` for plotting X-ray diffraction diagram. You should obtain something like.



task3. Crop the data in the [23, 24.5] degree range to focus on the first peak.

task4. Following the mathematical definition of a gaussian function :

$$g(x) = K + \frac{A}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

implement a function named `gauss(x,mu,sigma,A,K)` that takes x, μ, σ ,A and K as arguments and returns the values of $g(x)$.

task5. Use the `curve_fit` function that comes from the `scipy.optimize` module for computing best fit values of the x, μ, σ ,A and K values. From these results, deduce the position of the peak.

task6. Automatize this process for all the x-ray diffraction files and display x-ray diagrams on a same figure with the `subplot` function. In addition, you should display on each diagram :

1. the file name
2. the position angle of the first peak computed with the fitting function
3. vertical lines that highlights the peak positions

5.11 Playing with image

Goal	Use numpy for manipulating images
Prerequisite	language basics, numpy, matplotlib
Duration	1-hour
Correction	`cow.py`

The objective is to use numpy for manipulating images: color to gray scale conversion, crop, saturating, etc. We will play with the above image.

task1. Download the image file `cow.png` and place it on your current python working directory.

task2. Thanks to the `mpimg` module that comes from `matplotlib.image`, open the image and plot it with the `imshow` function.

At this step, you should have an `img` numpy array that describes the image in the Red Green Blue (RGB) image format.

task3. To learn more about the image and the RGB format, print the shape of the image array, the min and max values. You should obtain something like:

```
image shape    : (1137, 1820, 3)
image min value: 0.0
image max value: 1.0
```

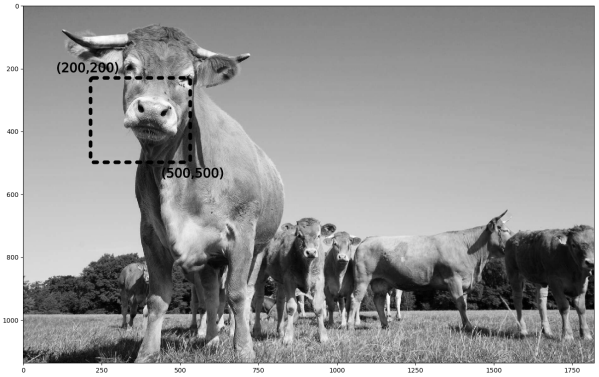
From the above information the image is 1137×1820 pixels and contains 3 channels related to the three colors: red, green and blue also named *channel*. `0` corresponds to no-color and `1` corresponds to the maximal color intensity. Each channel accepts values in the [0, 1] range. Now, we will move on black and white image. Black and white images are simpler because they contain only one channel: the gray scale channel in the [0,1] range. We will use the following function for grayscale conversion that takes in argument an rgb numpy array and returns a numpy array that describes the image in grayscale.

```
def rgb2gray(rgb_im):
    r, g, b = rgb_im[:,:,0], rgb_im[:,:,1], rgb_im[:,:,2]
    gray_im = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray_im
```

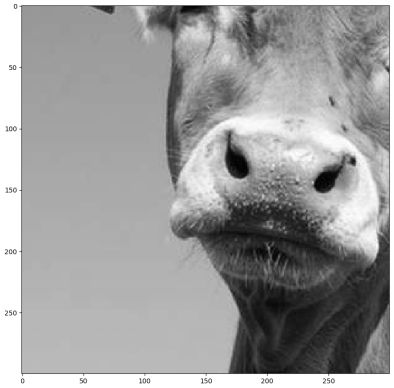
task4. Thanks to the `imshow` function, plot the related image converted in grayscale. You must use the `cmap=plt.get_cmap('gray')` optional argument to plot it in grayscale.

You should obtain something like this:

task5. Thanks to slicing, zoom on the area highlighted below.

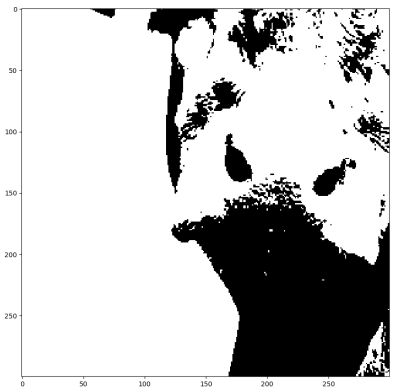


You should obtain something like this:



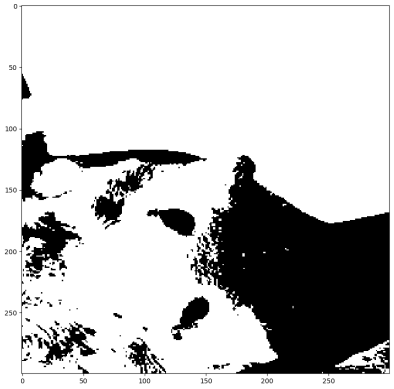
task6. Thanks to numpy filter, saturates the image. Pixels lower than 0.5 must take the zero value or one otherwise.

You should obtain something like this:



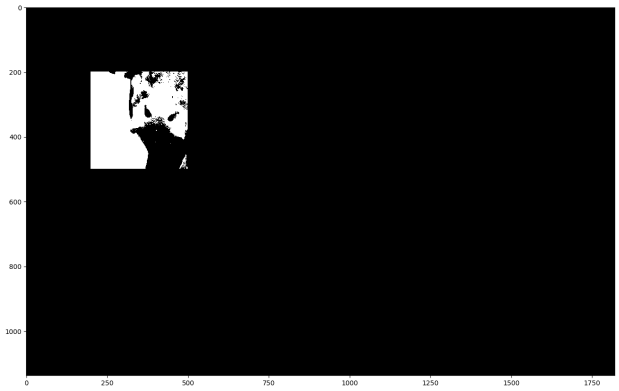
task7. Thanks to the numpy transpose function `.T`, make a 90° rotation

You should obtain something like this:



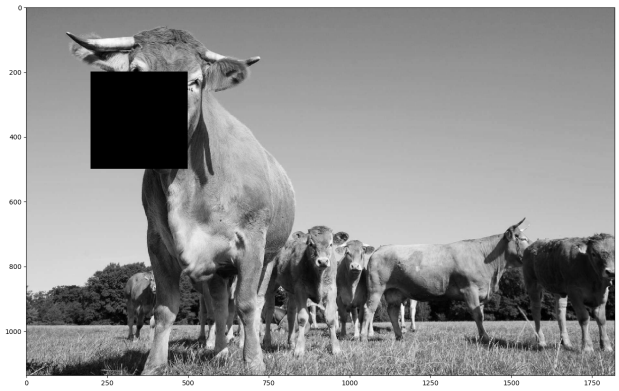
task8. Make a copy of the image and crop the working area.

You should obtain something like this:



task9. Finally, try to hide the working area of the image

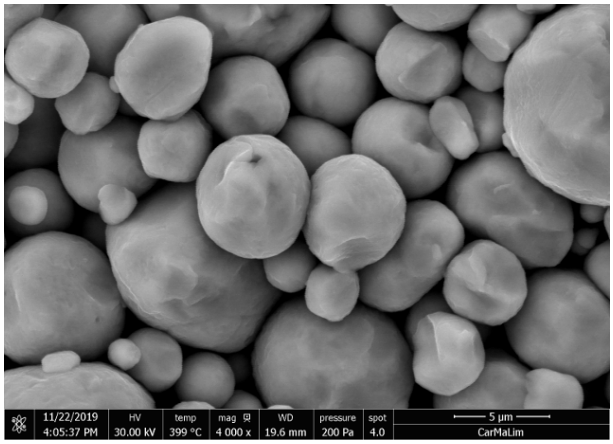
You should obtain something like this:



5.12 Finding contours of grains from SEM image

Goal	Use numpy for manipulating images
Prerequisite	language basics, numpy, image, matplotlib
Duration	1-hour
Correction	`grains.py`

The objective is to use numpy for facilitating object detection. Here the object to detect are grains that appear on a SEM image. The related SEM image is here.

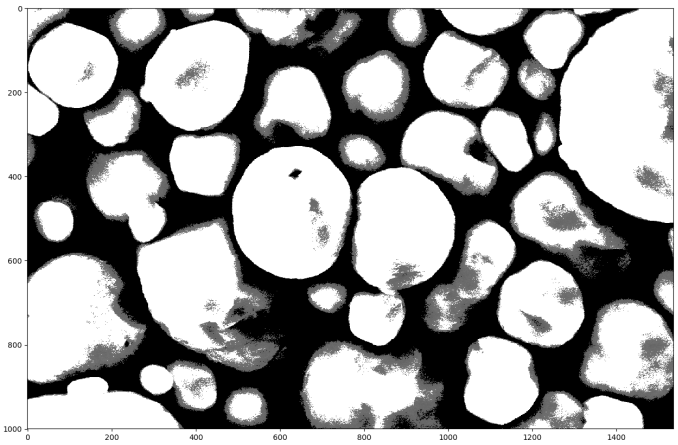


task1. Download the image file `grains.tif` and place it on your current python working directory.

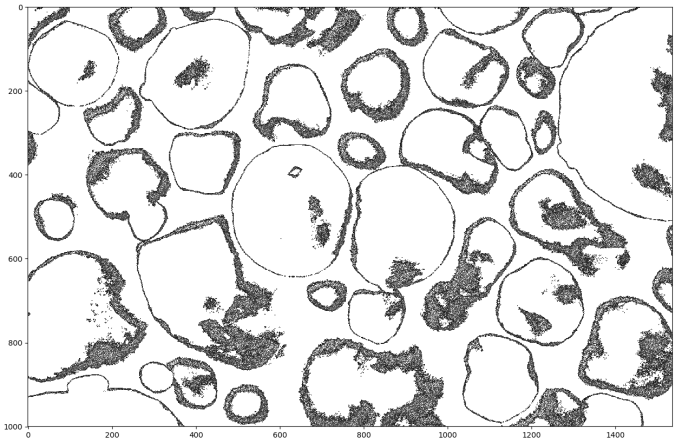
task2. As shown in the previous exercise, open the image with the `mpimg` module that comes from `matplotlib.image` and plot it with the `imshow` function.

task3. Use the slicing to crop the image for removing the information banner at the bottom of the image.

task4. Apply thresholds to highlight grains. You should obtain something like this :



task5. Apply the `np.gradient` to highlight grain contours. Apply a threshold and you should obtain something like this :


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

5.13 Ball tracking

Goal	
Prerequisite	language basics, numpy, scipy, matplotlib
Duration	3 hours
Correction	`magnus/magnus.py`

The objective of this practical exercise it to implement several layer of numerical treatment in order to track a moving object on a video. The original video is available here: `https://www.youtube.com/watch?v=r-i6XpcL1Fs`

task1. Download the image series `http://www.unilim.fr/pages_perso/damien.andre/cours/python/magnus-img.zip` and place it on your current python working directory.

task2. Thanks to the `glob` module, display one-by-one the name of the image files inside the `magnus-img` directory.

task3. Open each image with the `mpimg.imread()` function from the `matplotlib` module and convert them into gray-scale.

task4. Save the gray-scale images into another directory named `modified-img` thanks to the `scipy.misc.toimage()` function.

Now, let's play!

task5. Imagine several numerical treatments able to track the position (in pixel) of the falling ball.

You should obtain something like this video: `http://www.unilim.fr/pages_perso/damien.andre/cours/python/magnus-result.mp4`

5.14 A full Python tuner ♫

Goal	Detect the main note from a `*.wav` music file
Prerequisite	language basics, numpy, scipy, matplotlib
Duration	1-2 hours
Correction	`tuner/tuner.py`

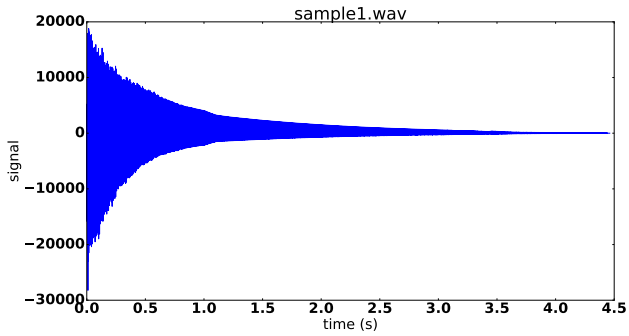
The objective of this training exercise it to use Python for detecting the main musical note of an instrument. To make it easier, the sounds are embedded in `*.wav` files. First, you need to download the following archive `http://www.unilim.fr/pages_perso/damien.andre/cours/python/wav-file.zip` that contains several music file in the raw `*.wav` format and unzip it.

task1. open one of these file thanks to the `wavfile` function that comes from the `scipy.io` module.

task2. put the collected signal into a numpy array. Note that you must manage two cases: stereo wave file or mono wave file.

task3. plot the obtained temporal signal versus time.

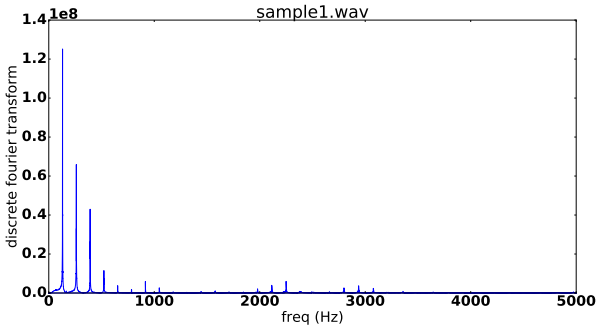
At this step, you must obtain something like this:



task4. thanks to the `np.fft.fft()` and `np.fft.fftfreq()` functions, make a spectral analysis of this temporal signal.

task5. compute the absolute value of the given fft and plot the spectral signal versus frequencies.

At this step, you must obtain the following chart:



task6. compute the main frequency of the spectral signal. It corresponds to the frequency where the fft is maximal. Note that you can use the `argmax()` method of numpy array,

Copy and paste the following code that corresponds to the frequencies of musical notes with their related names. If you experienced some problem with copy/paste, you can download directly this file from `tuner/notes.py`.

```
# frequencies of notes, these values were extracted
# from https://fr.wikipedia.org/wiki/Note_de_musique
note_f = np.array([32.70, 65.41,  130.81, 261.63, 523.25, 1046.50, 2093.00, 4186.01,
                   34.65, 69.30,  138.59, 277.18, 554.37, 1108.73, 2217.46, 4434.92,
                   36.71, 73.42,  146.83, 293.66, 587.33, 1174.66, 2349.32, 4698.64,
                   38.89, 77.78,  155.56, 311.13, 622.25, 1244.51, 2489.02, 4978.03,
                   41.20, 82.41,  164.81, 329.63, 659.26, 1318.51, 2637.02, 5274.04,
                   43.65, 87.31,  174.61, 349.23, 698.46, 1396.91, 2793.83, 5587.65,
                   46.25, 92.50,  185.00, 369.99, 739.99, 1479.98, 2959.96, 5919.91,
                   49.00, 98.00,  196.00, 392.00, 783.99, 1567.98, 3135.96, 6271.93,
                   51.91, 103.83, 207.65, 415.30, 830.61, 1661.22, 3322.44, 6644.88,
                   55.00, 110.00, 220.00, 440.00, 880.00, 1760.00, 3520.00, 7040.00,
                   58.27, 116.54, 233.08, 466.16, 932.33, 1864.66, 3729.31, 7458.62,
                   61.74, 123.47, 246.94, 493.88, 987.77, 1975.53, 3951.07, 7902.00])

# names of each note in the above array
note_id = ['C ', 'C ', 'C ', 'C ', 'C ', 'C ', 'C ', 'C ',
           'C#', 'C#', 'C#', 'C#', 'C#', 'C#', 'C#', 'C#',
           'D ', 'D ', 'D ', 'D ', 'D ', 'D ', 'D ', 'D ',
           'D#', 'D#', 'D#', 'D#', 'D#', 'D#', 'D#', 'D#',
           'E ', 'E ', 'E ', 'E ', 'E ', 'E ', 'E ', 'E ',
           'F ', 'F ', 'F ', 'F ', 'F ', 'F ', 'F ', 'F ',
           'F#', 'F#', 'F#', 'F#', 'F#', 'F#', 'F#', 'F#',
           'G ', 'G ', 'G ', 'G ', 'G ', 'G ', 'G ', 'G ',
           'G#', 'G#', 'G#', 'G#', 'G#', 'G#', 'G#', 'G#',
           'A ', 'A ', 'A ', 'A ', 'A ', 'A ', 'A ', 'A ',
           'A#', 'A#', 'A#', 'A#', 'A#', 'A#', 'A#', 'A#',
           'B ', 'B ', 'B ', 'B ', 'B ', 'B ', 'B ', 'B ']
```

task7. compute the index of the closest frequency (indexed by the `note_f` array) of the main frequency.

task8. deduce the name of the closest note. Thanks to the `print()` function, display the name of this note and the difference in Hertz from this note. The given output must looks like:

```
Analyzing 'sample1.wav'... Closest note is 'C ' with a difference of 0.16 Hz
```

This exercise is over, however, if you want to improve this program you may:

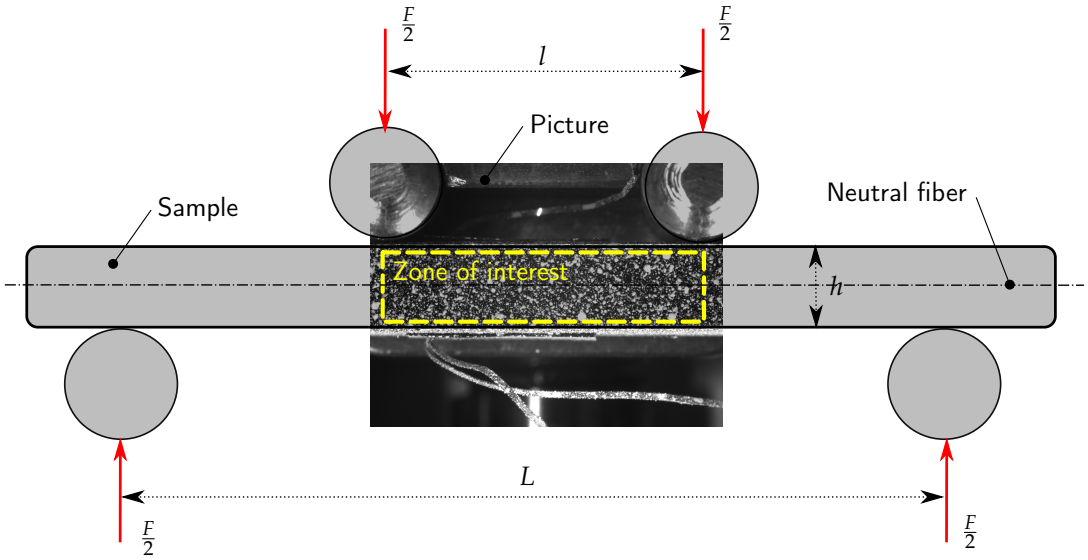
1. implement command line option with the `argparse` module to specify the name of the wav file at the execution of your script.
2. use the `pyaudio` module to record the sound through a microphone and for deducing on-the-fly the related note.

5.15 Post treating digital image correlation results

Goal	Computing Young's modulus from displacement field given by DIC
Prerequisite	language basics, material mechanics, images, numpy, scipy, matplotlib
Duration	2-4 hours
Correction	`dic/dic.py`

The aim of this exercise is to compute a material property (Young's modulus) from a Digital Image Correlation (DIC) processing. The DIC is an advanced numerical treatment able to retrieve a displacement field by comparing two images. Here, DIC is used on four point bending test.

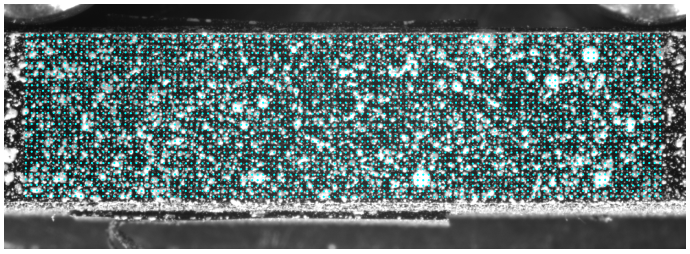
The image below shows the apparatus. The force F is applied through cylindrical rods. The sample is a rectangular beam rod with section lengths b and h. Image snapshots were taken during the test.



The different parameter values of the experiment are given in the following tab:

im	`(2560, 1920)`	image size	px
s	`6.1e-06`	image scale	$m.px^{-1}$
L	`36e-3`	distance of upper supports	m
l	`14e-3`	distance of lower supports	m
b	`7.66e-3`	sample width	m
h	`4.06e-3`	sample height	m
F	`400.`	the applied force	N
M_f	$\frac{F}{4} \times (L - l)$	the applied torque	$N.m$
I	$\frac{bh^3}{12}$	quadratic section momentum	m^4

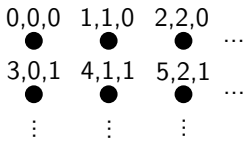
The images were analyzed with a DIC treatment. The following picture highlight this treatment. A point grid is introduced in the zone of interest. Finally, displacement of each point is computed thanks to DIC numerical treatments.



Download the file `dic/dic.csv` that contains the result of this DIC process. The file looks like:

```
index,index_x,index_y,pos_x,pos_y,disp_x,disp_y
0,0,0,103.0,604.0,0.939903259277,-2.02941894531
1,0,1,103.0,624.0,0.897819519043,-1.99945068359
2,0,2,103.0,644.0,0.843704223633,-2.00439453125
3,0,3,103.0,664.0,0.777610778809,-1.98834228516
.....
```

In the above file, columns are separated by comma characters. The first column is the global index of the data. The second and third columns are the x, y indexes of the related data. The following picture highlights the numeration convention: the first number is the global *index*, the second and third number are *index_x* and *index_y*.



The columns *pos_x* and *pos_y* give the coordinates in pixel of the related point. Finally, the *disp_x* and *disp_y* values are the displacement vector of this point.

task1. read this file and store the *pos_x*, *pos_y*, *disp_x* and *disp_y* values in two dimensional arrays where values are indexed by *index_x* and *index_y*. Note that the point grid size is 116×30. Your code must look like:

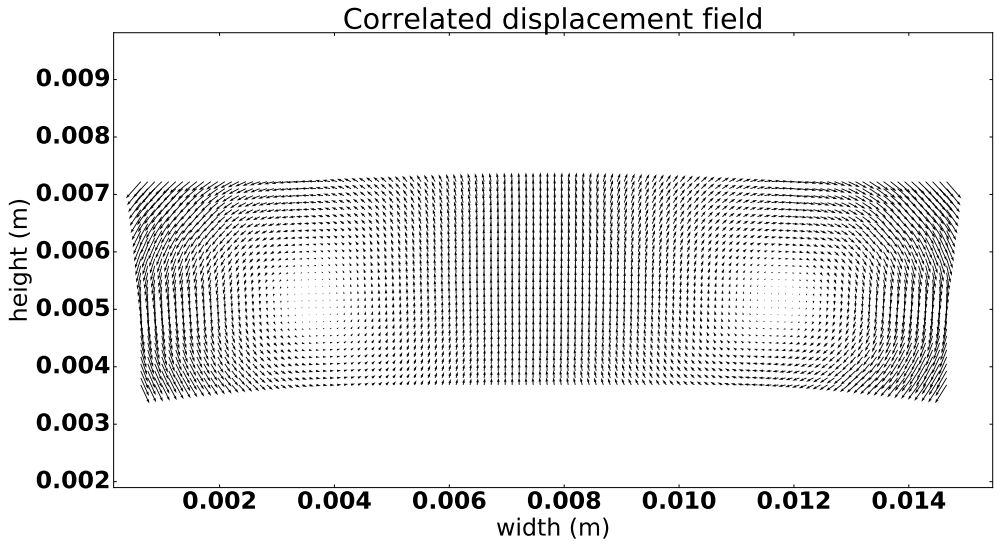
```
pos_x  = np.zeros((116, 30))
pos_y  = np.zeros((116, 30))
disp_x = np.zeros((116, 30))
disp_y = np.zeros((116, 30))
with open('./dic.csv', 'r') as f:
  # read file and fill arrays....
```

task2. translate these values from pixel to meter. The related scale factor is:

$$s = 6.1 \times 10^{-6} \text{ m.px}^{-1}$$

task3. Thanks to the matplotlib `quiver()` function, plots the displacement field with graphical arrows.

Your given plot must look like:



From the given displacement field, we want to compute the strain field. Remember that strain field is the gradient of the displacement field.

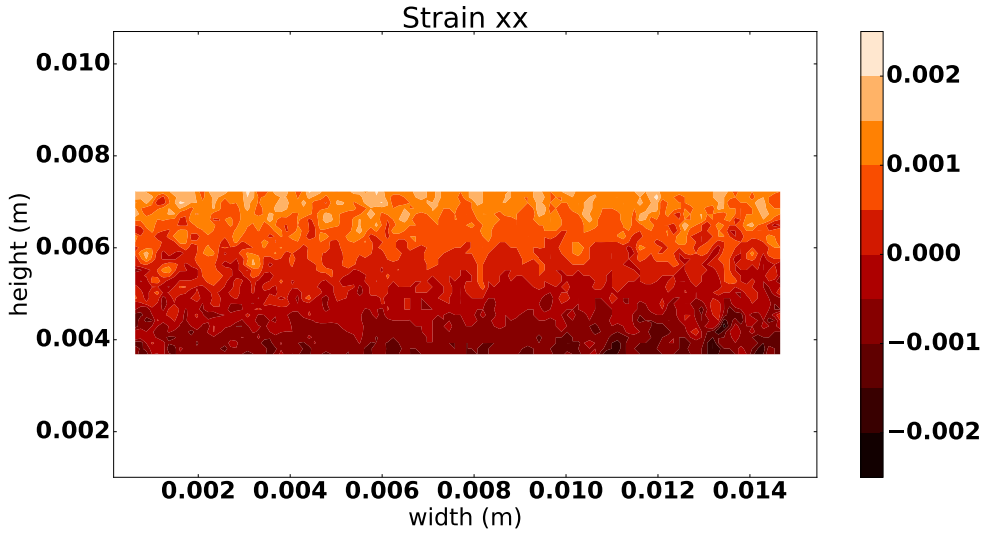
task4. thanks to the numpy `gradient()` function, computes the strain field ε_{xx}. The ε_{xx} strain is given by the relation:

$$\varepsilon_{xx} = \left[\vec{grad}\,(\vec{u}.\vec{x})\right].\vec{x} \tag{3}$$

where u is the displacement vector.

task5. thanks to the matplotlib `contourf()` function, plots the ε_{xx} strain field.

Your plot must look like:



Now, we want to compute the Young's modulus of this material. As you can see, the above plot is really noisy. In this condition, we are not able to compute the Young's modulus of this material from this raw data.

To compute the young's modulus, we propose to use the following relation that comes from

material strength theory:

$$E = -\frac{M_f}{f''(x) \times I} \tag{4}$$

where $f''(x)$ is the second derivative order of the deviation (displacement along y) of the neutral fiber of the beam.

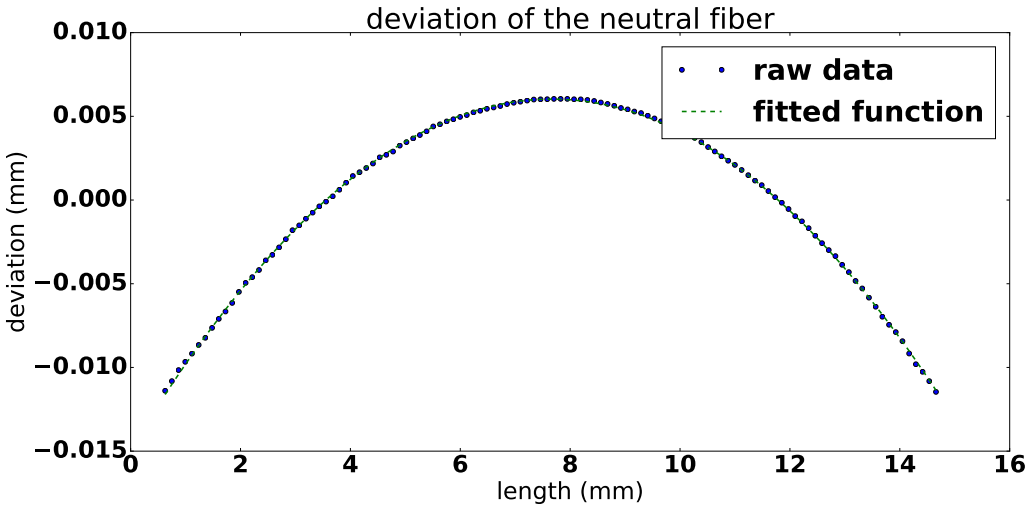
task6. from the displacement array along y (in the `disp_y` array) extract the deviation of the neutral fiber versus the position along x (in the `pos_x` array).

task7. thanks to the `curve_fit()` function that comes from `scipy.optimize`, compute a second order fitting function f of the deviation where f is:

$$f(x) = ax^2 + bx + c \tag{5}$$

task8. plot the raw data and the fitted function of deviation versus the position x.

Your plot must look like:



task9. thanks to equation 4, compute the Young's modulus of the material.

Compare your result with literature. The material tested here is a *duralumin* rod.

5.16 Discrete element from scratch

Goal	Build a mini 2D DEM program for compaction problem
Prerequisite	language basics, object oriented programming, dem method
Duration	4 hours
Correction	`dem/run-dem.py`

The aim of this exercise is to build from scratch a mini discrete element program able to study compaction problems. Before starting, you probably need to learn a little bit about Discrete Element Method (DEM).

task1. download the `dem/minidem.py` file, that contains some utility for DEM visualization and management.

task2. create a new python file and import the `minidem.py` file as a module.

task3. define a new class named `grain`. This class must have the following attributes: a radius, a mass, a position vector, a velocity vector and an acceleration vector. Note that you can use the `minidem.vec` class for vectors.

Note that the dimension of the scene is a square of 100x100 lengths.

task4. copy/paste the following code in your python file. Use it for building a discrete domain with 9x9 discrete elements aligned on a regular grid. The discrete elements must have a radius of 5 and a mass equal to 1.

```
  def time_loop():
    pass

dem.init()
dem.loop_function = time_loop
dem.max_iteration = 1000

# build your domain here

dem.run()
```

task5. thanks to the `random` module add some randomization for discrete element sizes and locations.

The time loop of a discrete element algorithm must contains the following mains steps. For all discrete elements:

1. set forces to zero,
2. add gravity forces ,
3. detect contact and add repulsive force if a contact is detected,
4. compute position at the next time step thanks to the velocity Verlet scheme and
5. apply boundary condition.

task6. implement the step 1, 2 and 4. Check your implementation by running your code. All the discrete element must fall down.

task7. apply boundary conditions for constraining grains to stay inside the 100x100 scene. You can apply a reflection law with a *coefficient of restitution* around 0.9. It means that a discrete element will have a velocity 10% lower after a collision with a wall.

task8. build a new `collide` function that takes two discrete elements as argument. If a collision is detected between the two discrete elements, the `collide` function computes and applies repulsive forces on both discrete elements. The repulsive force $\mathbf{f}$ must be:

$$\mathbf{f} = K.\delta.\mathbf{n}$$

where K is the contact stiffness (around 10,000), δ is the interpenetration and $\mathbf{n}$ is the contact normal.

task9. in order to stabilize the simulation, add a damping force to the repulsive force in the `collide` function. The damping force $\mathbf{f_d}$ is:

$$\mathbf{f_d} = d.\delta_v.\mathbf{n} \tag{6}$$

where d is the damping factor (around 14.1), δ_v is the norm of the velocity difference between the two discrete elements and $\mathbf{n}$ is the contact normal.

Now, you can play with a minimal discrete element code able to study the 2D packing of arbitrary disks. Note that this problem is an open problem of the granular media science. At this time, it can not be solved by analytic laws!

6 Conclusion

Now, you have the basics of Python programming for scientific stuffs. With only basics, you are able to do sophisticated things such as command line software or advanced numerical treatments and computation.

You probably want to build nice Graphical User Interface (GUI) with Python. Be aware, my advice is: do a GUI only if this is really required. Programming GUI, even in Python, is a very fastidious and frustrating stuff. Graphical programs are more than ten times longer that command line programs for doing the same job. Therefore, if you really need GUI, you can choose several packages such as `tkinter` or `pyQt`.

The next part of this document will focus on usage of Python for studying and solving a special field. So, let's move on a very interesting topic concerning Artificial Intelligence (AI).

Never forget forget that **programming is fun**!

Part 2

Machine learning with Python

1 Introduction

1.1 Artificial Intelligence... really ?

Today, a lot of noise is generated concerning Artificial Intelligence (AI) and machine learning. To help us decreasing this noise, we will study, from scratch, the fundamental of neural network. Be aware, I am not a specialist of this topic. As numerical scientist enthusiast, I am curious about IA. That's why, I spent some times for working on this subject in order to get some understanding about how IA works. So, this part is only an introduction to neural networks. In fact, I will share with you what I have (approximately) understood.

Neural network is a special class of AI technology. Today, *convolutional* neural network seems to be one of the most advanced IA techno. It allows *deep learning* able to do very impressive things. In my opinion, one of the most impressive thing that I found, is the usage of deep learning for art generating. For example, *deep dream* [2] is an AI able to generate painting art by combining photos and human paintings. The results are really amazing!

Indeed, IA algorithms need an impressive amount of data to be efficient. In fact, AI is not a correct name because it gives a false idea of how AI works and what AI can do. A more precise name is *machine learning algorithm*.

Okay, let's go... However, before beginning, let's clarifying the purpose of AI.

1.2 The purpose of machine learning (and dinosaur)

The aim of a neural network is to make *predictions*. If you are a scientist, you probably know the linear regression concept. This mathematical tool allows us to make predictions. Imagine that your are a paleontologist. You have collected several dinosaur bones and you put them into the table below.

dino		A	B	C	D	E
femur length (cm)	f	38	56	59	64	74
humerus length (cm)	h	41	63	70	72	84

Now, you want to know if the dino A, B, C, D and E belong to the same species (a class in fact). As you probably know, the size of an individual depends on both genetic factors and ages. So, let's do the assumption that a linear factor K exists between the femur f and humerus h sizes such as:

$$h = K \times f$$

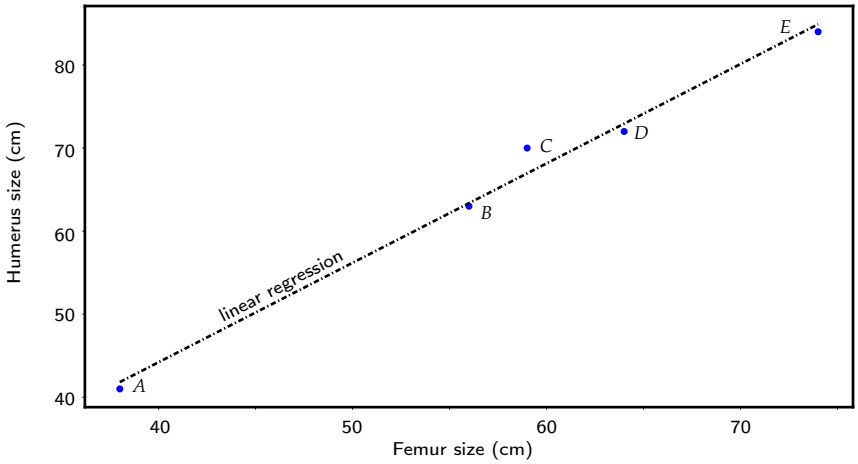
and the value of this factor K is supposed to be a characteristic of a given dinosaur species. Now, let's plot these data and the associated linear regression... with python of course :)!

```
from matplotlib import pyplot as plt
import numpy as np
x = [38, 56 , 59 , 64 , 74]
y = [41 ,63 , 70 , 72 , 84]
```

[2] https://deepdreamgenerator.com/

```
fit = np.polyfit(x,y,1) # linear regression here
fit_fn = np.poly1d(fit) # put it as lambda expression
plt.plot(x,y, 'bo', x, fit_fn(x), '--k')
plt.xlabel('Femur size (cm)')
plt.ylabel('Humerus size (cm)')
plt.show()
```

It gives the following plot:



On this plot, it is clear that some doubts exist on the dino C because the point C is far from the linear regression line. So, we can suppose that the dino C may not belong to the same species than other dino.

In fact, we do here a *binary classification*: we try to deduce if a point belongs to a certain class or not. This kind of problem can be easily tackled with the most simple neural network. We will study this neural network in details in the next section.

2 Single layer neural network (Rosenblatt's Perceptron)

The previous example can be tackled manually because the data sample is small. Indeed, if you want to automatize this process for a large amount of data, you need to automatize it with a program. In 1957, The Rosenblatt's perceptron [**?**] was the first learning machine. As shown on picture 5, it was really a machine!

Here, we will see the most simple example of *supervised training*. It means that the machine will be trained with input data (noted $\mathbf{x}$) that has already been classified (noted $\mathbf{t}$). For example, the next figure shows an example of already-classified data. Here, for the sake of clarity, the data $\mathbf{x}$ are in two dimensions. Each point of the data is defined by 2D coordinates (x_0, x_1) and each point is associated with a class. The class is defined by a label t that takes the '1' or '0' value.

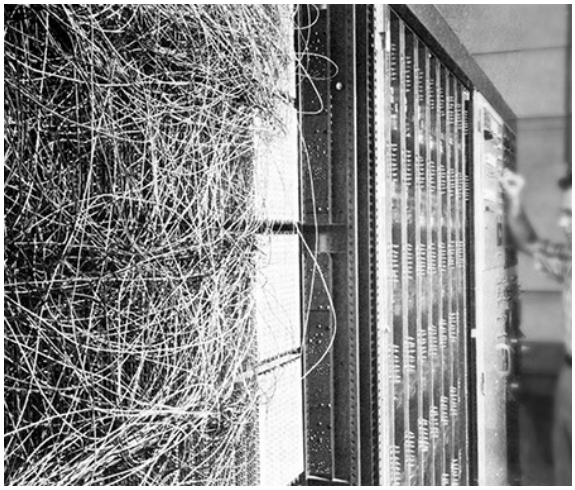
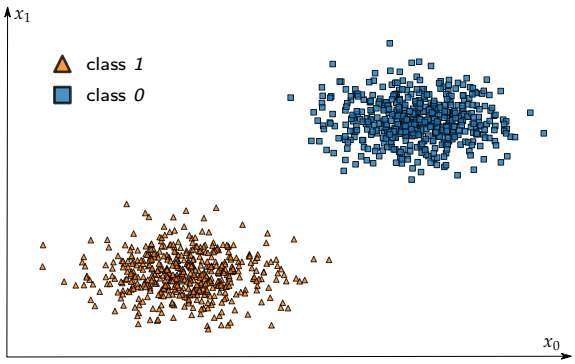


Figure 4: The Rosenblatt's perceptron machine in 1957



The workflow of the supervised machine learning is simple:

1. training the machine with already-classified data and
2. make prediction with non-classified data.

Figure 5 describes the Rosenblatt's Perceptron architecture. The output is generated from the input through two functions: the *transfer function* and the *activation function*. The weights w_i are the coefficients that must be trained and adjusted during the learning step.

The goal of the training step is to deduce the best weight values from already-classified data. In the original Perceptron, these weights was generated through mechanical potentiometers and their values were automatically adjusted with motors!

2.1 The feed forward rule

The *feed forward* rule is the rule that makes output value(s) from input values. It induces different mathematical processes and computations. In the Perceptron algorithm, two computations are processed given by two different functions: the *transfer* function and the *activation* function.

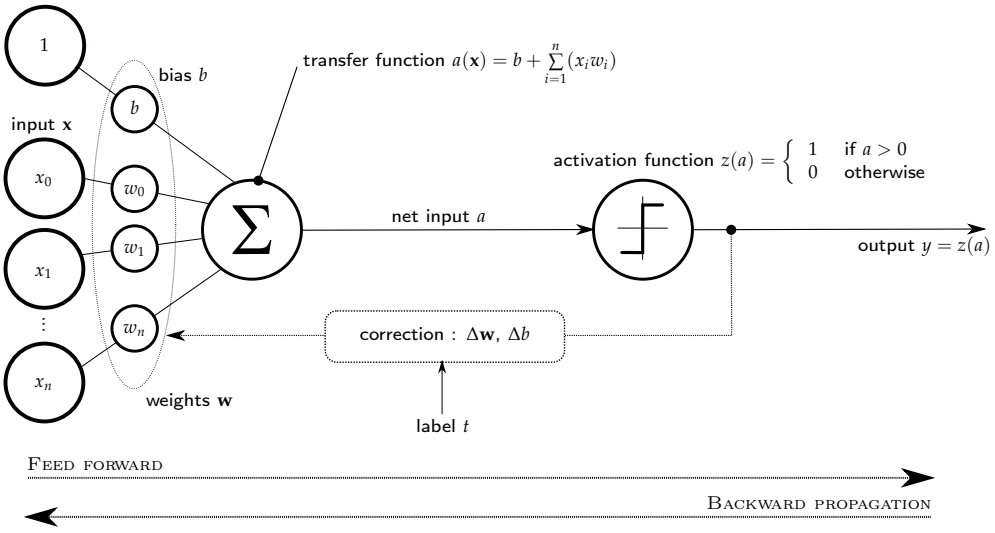


Figure 5: Overview of the Perceptron architecture

2.1.1 The transfer function

The *transfer function* makes a linear combination of the inputs $(x_0, x_1, \ldots, x_n)$ with the weights $(w_0, w_1, \ldots, w_n)$ and a single coefficient b named *bias*. The transfer function is:

$$a(x_i) = b + x_0 w_0 + x_1 w_1 + \cdots + x_n w_n = b + \sum_{i=1}^{n}(x_i w_i)$$

We can simplify the mathematical written by expressing inputs $\mathbf{x}$ and weights $\mathbf{w}$ with vectors:

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$

Now, the transfer function a can be written as:

$$a(\mathbf{x}, \mathbf{w}, b) = \mathbf{x} \cdot \mathbf{w} + b \tag{7}$$

where $\cdot$ is the scalar (or dot) product between two vectors.

Note that the transfer function gives a scalar a. This function is **really important**. It allows us to make linear combination of entries. This function is already used even in complex modern neural networks such as convolutional neural networks used in deep learning.

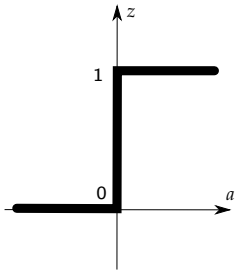
2.1.2 The activation function

The second function is called *activation function*. This is always a non-linear function. In our case (the original Perceptron), the activation function is the output of the Perceptron and must be able to separate the two classes with `'1'` nor `'0'` values. The *unit step function* can do this job.

The unit step function is defined as:

$$z(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

Graphically, the unit step function gives:



2.2 The backward propagation rule

At this step, we are able to compute an output y that takes `'1'` nor `'0'` values from an input $\mathbf{x}$. However, the weights $\mathbf{w}$ has not been trained and the computed output is irrelevant. The backward propagation rule allows to deduce the best fit weights from already-classified data.

The rule proposed by Rosenblatt is quit simple. For each input data $\mathbf{x}$, the weight $\mathbf{w}$ and bias b are adjusted with a small correction $\mathbf{\Delta w}$ and Δb as follows:

$$\begin{aligned} \mathbf{w} &\to \mathbf{w} + \eta \mathbf{\Delta w} \\ b &\to b + \eta \Delta b \end{aligned} \tag{8}$$

where η is a coefficient named the *learning rate*. Note that η is a scalar. Rosenblatt has proposed to deduce these corrections from the following rules:

$$\begin{aligned} \Delta w_0 &= \eta\,(t-y)\,x_0 \\ \Delta w_1 &= \eta\,(t-y)\,x_1 \\ &\vdots \\ \Delta w_n &= \eta\,(t-y)\,x_n \\ \Delta b &= \eta\,(t-y) \end{aligned}$$

where t is the target value and y is the output of the Perceptron. The above formula can be written in vectorial form as:

$$\begin{aligned} \mathbf{\Delta w} &= \eta\,(t-y)\,\mathbf{x} \\ \Delta b &= \eta\,(t-y) \end{aligned}$$

With this learning rule, if $t = y$ the weights are not modified. If $t \neq y$ the weight values are adjusted. This trick allows to increase or decrease the weights in the direction of the target value. The rate of these modifications can be adjusted thanks to the η coefficient.

2.3 Let's coding!

In addition of *matplotlib* and *numpy*, we will use the `scikit-learn` [3] and the `mlxtend` [4] modules for generating and visualizing data. The following python code generates pseudo-randomized data thanks to the `make_blobs(...)` function that comes from `sklearn` module. These data are finally plotted with the `plot_decision_regions(...)` function that comes from `mlxtend` module.

Listing 1: perceptron-data.py

```
import numpy as np
from sklearn import datasets
from matplotlib import pyplot as plt
from mlxtend.plotting import plot_decision_regions

# make pseudo randomize data
data,label = datasets.make_blobs(n_samples=1000, centers=2, n_features=2)

# the perceptron class, this is mandatory for plotting with mlxtend
class Perceptron:
    @staticmethod
    def predict(X):
        return np.zeros(len(X))

# plotting
plt.figure()
plot_decision_regions(data, label, clf=Perceptron)
plt.show()
```

In the above code, data are generated with the `make_blobs(...)` function. This function returns two parameters. Here these parameters are taken as variables named `data` and `label`. Let's monitor these variables with the interpreter.

```
>>> type(data)
<type 'numpy.ndarray'>
>>> data.shape
(1000, 2)
```

> *If you want to stop the execution of a python script at a given step of your code, you just have to insert the `code.interact(local=locals())` function that comes from the `code` module.*

Here, you can see that `data` is a numpy array. The first axis length (1000) corresponds to the number of points specified with the `n_samples` argument of the `make_blobs(...)` function. The second axis length corresponds to the (2D) coordinates of each point that corresponds to the `n_features` argument of the `make_blobs(...)` function. Here, we have chosen two dimensions for an easy plotting of data. To summarize, `data` is simply an array that contains 1,000 two dimensional data points.

In addition, these 1,000 data points are classified through the `label` variable.

```
>>> type(label)
<type 'numpy.ndarray'>
>>> label.shape
(1000,)
```

[3] https://scikit-learn.org/stable/
[4] http://rasbt.github.io/mlxtend/

As you can see, the `label` variable contains the class (also called label) of each 1,000 points. Here, it contains only two classes identified by '0' or '1'. You can choose more classes with the `centers` argument of the `make_blobs(...)` function.

```
>>> data
array([[-2.69560892, -1.72976383],
       [-2.12983421, -1.78565134],
       [-0.71161969,  8.43909972],
       ...,
       [-1.31913952, -4.69944969],
       [-4.46749849, -4.19619198],
       [-1.83938485,  0.02932308]])
        x0            x1
```

1,000 data points

```
>>> label
array([0,
       0,
       1,
       ...,
       0,
       0,
       0])
        t
```

1,000 data labels'

The `make_blobs(...)` function, that comes from the `sklearn` module, generates data cloud (a blob) centered at a given location as shown on Figure 6. This kind of data are *linearly separable*. It means that a straight line (a linear function) is able to separate (or classify) these kind of data. Note that this property can be generalized for n-dimensional spaces with hyperplanes.

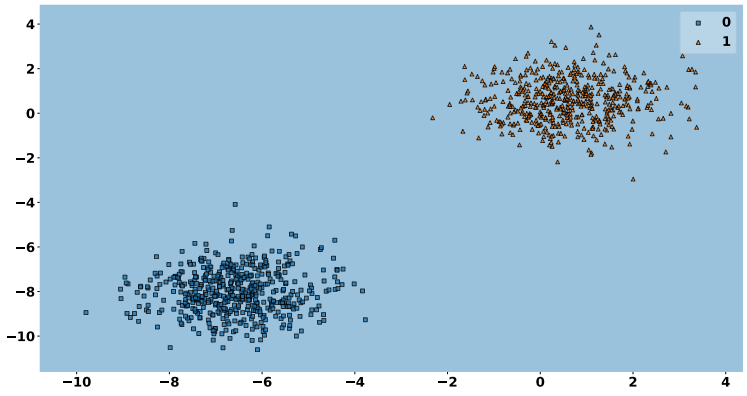


Figure 6: Example of labeled data generated with the `make_blobs(...)` function

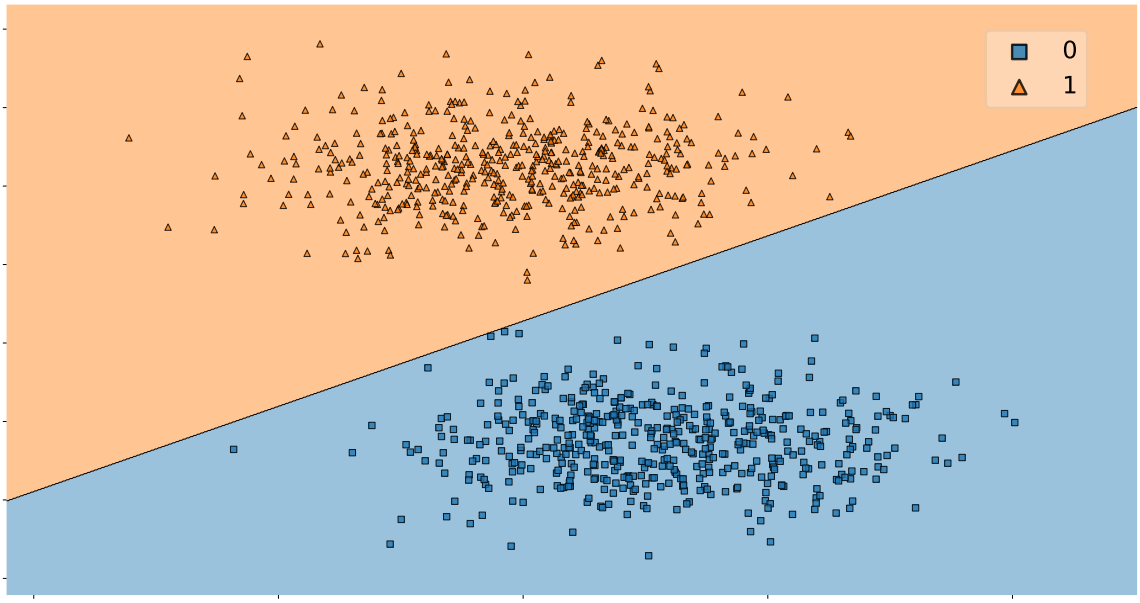
Now, we can apply the recipes given in sections 2.1.1 and 2.1.2 in order to implement the Perceptron. It gives the following code.

Listing 2: `perceptron-init.py`

```
import numpy as np
from sklearn import datasets
from matplotlib import pyplot as plt
from mlxtend.plotting import plot_decision_regions

# make data
data,label = datasets.make_blobs(n_samples=1000, centers=2, n_features=2)

# initialize paramaters
w = np.zeros(2) # it contains (w0,w1)
b = 0.
eta = 0.001 # learning rate

# let's compute, we iterate over all the date
for x,t in zip(data,label):
    # feed forward
    a = np.dot(x, w) + b
```

```
    z = 1 if a>0. else 0
    y = z
    #backward propagation
    w += eta*(t-y)*x
    b += eta*(t-y)

# the perceptron class, this is mandatory for plotting with mlxtend
class Perceptron:
    @staticmethod
    def predict(X): # here, X is a numpy array that contains all the data
        a = np.dot(X, w) + b
        return np.where(a >= 0., 1, 0)

# plotting
plt.figure()
plot_decision_regions(data, label, clf=Perceptron)
plt.show()
```

The image below plots the results of the above code. As you can observe, the Perceptron is able to separate the variable with a straight line. But.... how it really works?



Take a look at equation 7. We can write this equation related to the net *input function* with 2 dimensional data(x_0, x_1) as follow:

$$a = x_0 w_0 + x_1 w_1 + b$$

Now, remember that the *activation function* $z(a)$ (see section 2.1.2) consists in classifying data depending on the sign of the input a :

$$z(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a < 0 \end{cases}$$

Here, we can take the frontier limit by posing a nor negative and nor positive, i.e, $a = 0$. It gives:

$$0 = x_0 w_0 + x_1 w_1 + b$$

Now, we can express x_1 as a function of x_0 as:

$$x_1 = -x_0 \frac{w_0}{w_1} - \frac{b}{w_1} \tag{9}$$

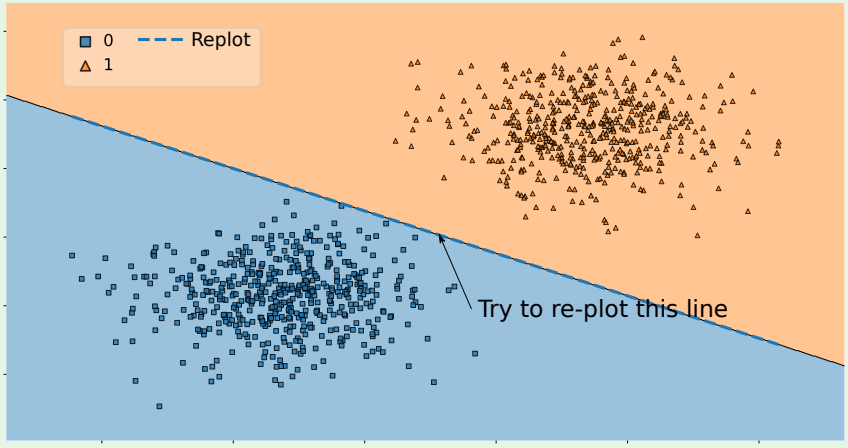
This last equation is linear, we retrieve the well known form $y = mx + y_0$ where:

$$m = -\frac{w_0}{w_1} \quad \text{and} \quad y_0 = -\frac{b}{w_1}$$

So, if the values of the coefficients w_0, w_1 and b are known, we are able to plot the function $x_1 = f(x_0)$. The related function is linear and the related plot is a straight line. This straight line separates the space in two half spaces:

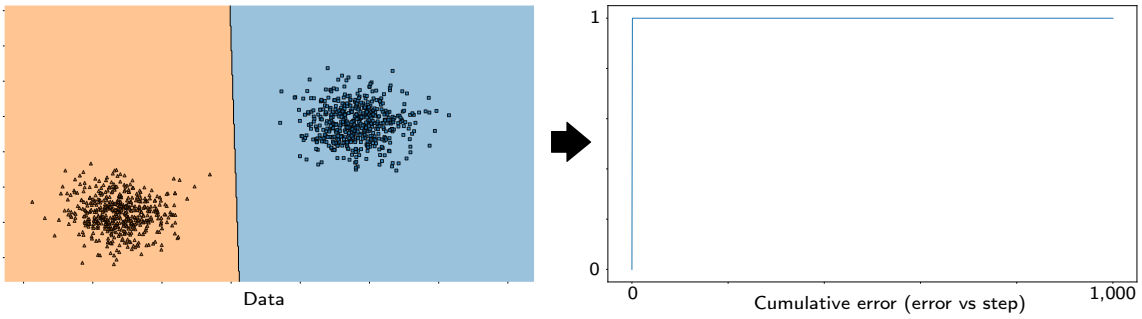
1. the half space below the line corresponds to the case where $a > 0$, e.g, the `'1'` class and
2. the half space above the line corresponds to the case where $a < 0$, e.g, the `'0'` class.

☞ *To verify the above assumption, you can try to plot the related equation 9. You must obtain something like this (dash dot line named Replot).*



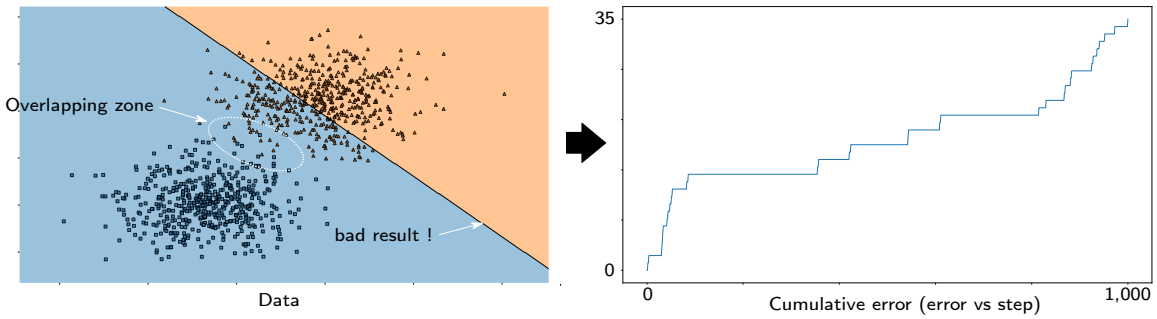
2.4 Converging rate, error and epoch

Let's interested us to the error that appears during the learning process and how the Perceptron algorithm converges toward a solution. The graph below plots the data and the cumulative errors made during the learning process. *Cumulative errors* means that each time the Perceptron gives the wrong answer during the learning process, the related error increments its value of one. Here, the cumulative error is a measure of the learning process rate of the machine.

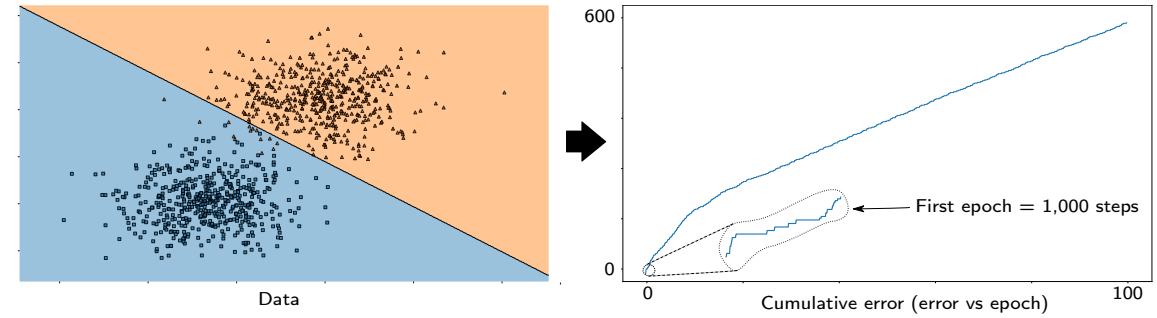


In this above case, the Perceptron made only one mistake on the whole 1,000 data sample and

the result looks really good. In fact, this good result is given because data are easily linearly separable. Let's see a more complicated example with an overlapping zone between the two classes as shown below.



In this new sample, an overlapping zone exists and the data are not clearly linearly separable. As, you can see, the results given by the Perceptron is bad because the separated line is not well placed. This bad result is also highlighted on the cumulative errors evolution that rises up continuously. In order to minimize this error, let's try to pass again the whole data sample into the learning process. The number of loop where the whole data sample are processed in the learning process is called **epoch**. The following charts show the results for 100 epochs!



As you can see, the results is quiet better: the frontier between the two classes looks better than the previous one.

This data set are available at the following urls:
1:perceptron-x.dat
2:perceptron-y.dat
3:perceptron-label.dat
If you want to use this data set, please download these three files, put them into your current directory and add the following lines to your script file:

```
# load the data from files
x  = np.fromfile('perceptron-x.dat')
y  = np.fromfile('perceptron-y.dat')
label = np.fromfile('perceptron-label.dat', dtype=int)
data = np.stack((x, y), axis=-1)
```

So, the epoch trick gives here better results. The listing below shows the implementation of the Perceptron with epochs.

Listing 3: perceptron-error.py

```
# ... same as before
epoch = 100 # epoch number
err   = [0] # to monitor the error

# let's compute, we iterate over all the date
for i in range(epoch): # ADD A LOOP HERE FOR EPOCH!
    for x,t in zip(data,label):
        # feed forward
        a = np.dot(x, w) + b
        z = 1 if a>0. else 0
        y = z
        #backward propagation
        w += eta*(t-y)*x
        b += eta*(t-y)
        # monitor error
        err.append(err[-1] + np.abs(t-y))
```

However, as you can see on the previous graphs, the cumulative error seems to linearly increase and do not stabilize toward a single value. This is normal, because as you can see on the previous figure some points are misclassified. It generates single errors that accumulate and induce a regular increase of the cumulative error curve. In fact, in this special case, the data are not fully linearly separable. That's why these errors are unavoidable.

Indeed, a better approach consists in generating a score between the $[0, 1]$ range that allows us to generate a probability that a point belongs to one of these two classes.

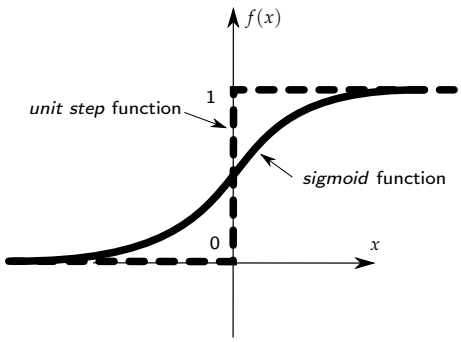
- A score closes to 0 means that a point as a high probability to belong to the `'0'` class.
- A score closes to 1 means that a point as a high probability to belong to the `'1'` class.

The unit step function can not do this job because it only gives two discrete values: 0 or 1. A good function to do this job, is the *sigmoid function* described in the next section.

3 Sophisticated backpropagation step with gradient descent

3.1 Implementing the sigmoid function

The following figure shows the main difference between the unit step function and the *sigmoid* function.



The main differences between these two functions are:

1. the unit step function is a discrete function which gives only 0 or 1 values from $\mathbb{R}$ and
2. the sigmoid function is a continuous non-linear function which gives any values in the $[0, 1]$ range from $\mathbb{R}$.

The unit step function is *non-derivable* whereas sigmoid function is *derivable*. The equation of the sigmoid function is given by:

$$\sigma(x) = \frac{1}{1+e^x}$$

and its derivative is given by:

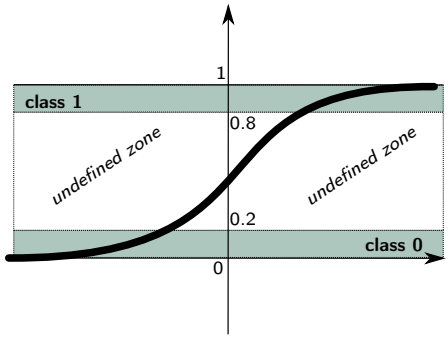
$$\sigma'(x) = \sigma(x)\ (1-\sigma(x))$$

Using sigmoid as activation function can be viewed as introducing probability in our machine:

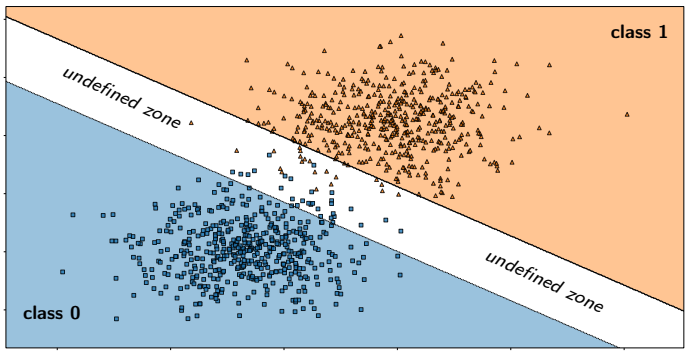
1. if the sigmoid function gives a value close to 1, it means that the related point has a strong probability to belong to the `'1'` class and
2. if the sigmoid function gives a value close to 0, it means that the related point has a strong probability to belong to the `'0'` class,
3. otherwise the point can not be defined and a high uncertainty remains on its class.

A simple approach consists in defining thresholds in order to classify input. For example, as shown on the figure below, we consider a threshold of 20%. It means that:

1. if the score is lower than 0.2, the point is considered as belonging to the `'0'` class,
2. if the score is higher than 0.8, the point is considered as belonging to the `'1'` class,
3. otherwise the point class is considered as uncertain.



This approach allows us to make three different zones. With the previous data set, it gives the following chart.



The next listing shows the implementation of the sigmoid function in the Perceptron.

Listing 4: perceptron-sigmoid.py

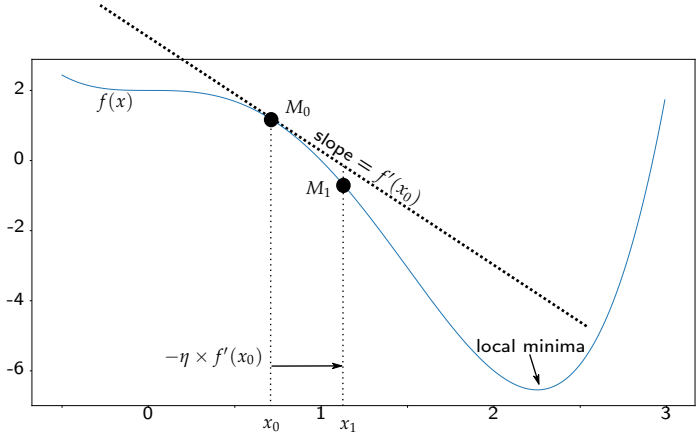
```
# ... same as before
def sigmoid(x): return 1 / (1 + np.exp(-x))

for i in range(epoch):
```

```
    # let's compute, we iterate over all the date
    for x,t in zip(data,label):
        # feed forward
        a = np.dot(x, w) + b
        z = sigmoid(a)
        y = z
        # backward propagation
        w += eta*(t-y)*x
        b += eta*(t-y)
        # monitor error
        err.append(err[-1] + np.abs(t-y))

# the perceptron class, this is mandatory for plotting with mlxtend
class Perceptron:
    @staticmethod
    def predict(X): # here, X is a numpy array that contains all the data
        a = np.dot(X, w) + b
        z = sigmoid(a)
        # it returns :
          #  0 if z <= 0.2
          #  1 if z >= 0.8
          # -1 elsewhere
        return np.where(z >= 0.8, 1, (np.where(z<=0.2, 0, -1)))
```

Another advantage of using this kind of continuous function comes from that the sigmoid function is fully **derivable**. This property can be used advantageously for optimizing the backward propagation process with the *gradient descent* algorithm and loss functions. The next section will introduce this highly important concept in ML: gradient descent algorithm.

3.2 The gradient descent algorithm

As illustrated in two dimensions on the next figure, the gradient descent algorithm is an iterative algorithm able to find the minimum of a derivable function $f(x)$.



It computes, from a starting point M_0 that belongs to $f(x)$ (the function to find the minimum), a next point by *descending* the tangent. The tangent at M_0 is given by the first order derivative of the function $f'(x_0)$. This process is repeated until the algorithm converge to a single point that means that the algorithm find a *local minima*. In a similar way of the Perceptron, the convergence rate can be adjusted with a coefficient noted η. The algorithm is quit simple and can be described with the following recursive formula:

$$x_{k+1} = x_k - \eta f'(x_k) \tag{10}$$

where k is the step number of the computation and x_k is the X axis value at the kth step of the algorithm. The figure 7 shows the converging rate of the gradient descent algorithm. In this example, the function $f(x)$ is:

$$f(x) = x^4 - 3x^3 + 2 \quad \text{and} \quad f'(x) = 4x^3 - 9x^2$$

On this plot, the orange points highlight the different steps of the algorithm. It suggests that the points roll down on the curve! In fact, the gradient descent algorithm allows to move the next point in a given *direction* and an *intensity*. In 2D, only two directions are possible: move to the left or move to the right whereas the intensity is related to the step between two consecutive points. You can note that this step is not constant (in contrast of the original Perceptron algorithm) and it decreases while approaching the local minima. It means that the recursive algorithm works well and converges toward the right minima value. The related python code is given in the next listing.

Listing 5: gradient-descent.py

```
cur_x     = 0.2     # where the algorithm starts
eta       = 0.01    # step size multiplier
precision = 0.00001 # the precision to achieve
previous_step_size = 1

xx   = [] # to plot results

f  = lambda x: x**4 - 3*x**3 + 2
df = lambda x: 4 * x**3 - 9 * x**2

while previous_step_size > precision:
    prev_x = cur_x
    cur_x -= eta * df(prev_x)
    previous_step_size = abs(cur_x - prev_x)
    xx.append(cur_x)

print("The local minimum occurs at", cur_x)
```

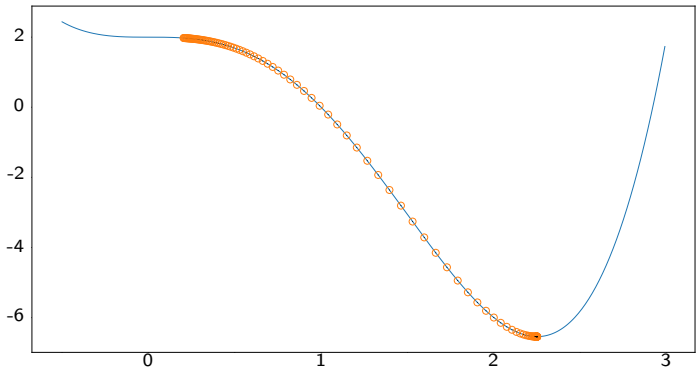


Figure 7: 2D gradient descent algorithm in action for $f(x) = x^4 - 3x^3 + 2$

However, one notes that the algorithm can be trapped in a local minima that do not correspond to the global minima of the function. For example, it happens in the given example if you start the algorithm at $x_0 = -0.5$. A second observation concerns the convergence rate η. The algorithm may diverge if η is too high. So, small η values ensure convergence but the number of iteration to compute becomes high and the algorithm slows down.

The gradient descent algorithm can also be applied in a 3 dimensional space. Imagine that the

function f depends on two variables x and y. So, $f(x,y)$ can be plotted as a surface in a three dimensional space. In a 3 dimensional space, the gradient descent algorithm becomes:

$$(x_{k+1},\ y_{k+1}) = (x_k,\ y_k) - \eta \nabla f(x_k,\ y_k)$$

where ∇ is the gradient operator:

$$\nabla f(x,\ y) = \left(\frac{\partial f}{\partial x},\ \frac{\partial f}{\partial y}\right)$$

Let's try it on a practical example. Suppose the following function (see figure 8):

$$f(x,y) = \sin\left(0.1\ x^2 + 0.05\ y^2 + y\right)$$

Here, the partial derivatives required for computing the gradient $\nabla f(x,\ y)$ are:

$$\frac{\partial f}{\partial x}(x,y) = \cos\left(0.1\ x^2 + 0.05\ y^2 + y\right)(0.1\ x)$$
$$\frac{\partial f}{\partial y}(x,y) = \cos\left(0.1\ x^2 + 0.05\ y^2 + y\right)(0.05\ y + 1)$$

From the above formula, the gradient descent algorithm can be implemented. The next listing shows its implementation.

Listing 6: gradient-descent-3D.py

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np


f     = lambda x,y: np.sin(0.1*x**2 + 0.05*y**2 + y)
df_dx = lambda x,y: np.cos(0.1*x**2 + 0.05*y**2 + y)*(0.2*x)
df_dy = lambda x,y: np.cos(0.1*x**2 + 0.05*y**2 + y)*(0.1*y+1)

cur_x = -4. # The algorithm starts
cur_y = -.5 # The algorithm starts
gamma = 0.1 # rate
precision = 0.000001
previous_step_size = 1

# for plotting
xx   = []
yy   = []
while np.all(previous_step_size) > precision:
    prev_x = cur_x
    prev_y = cur_y
    cur_x -= gamma * df_dx(prev_x, prev_y)
    cur_y -= gamma * df_dy(prev_x, prev_y)
    previous_step_size = np.array([abs(cur_x - prev_x), abs(cur_y - prev_y)])

    xx.append(cur_x)
    yy.append(cur_y)

# Make data.
X = np.arange(-5, 5, 0.02)
Y = np.arange(-5, 0, 0.02)
X, Y = np.meshgrid(X, Y)

# plot data
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')
plt.axis('equal')
ax.plot_wireframe(X, Y, f(X,Y), rstride=10, cstride=10,alpha=0.5)
ax.plot_surface(X, Y, f(X,Y), cmap=cm.coolwarm, linewidth=0, antialiased=False, alpha=0.2)
ax.scatter(xx, yy, f(np.array(xx),np.array(yy)), color='red')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

This listing gives the figure 8 that shows the gradient descent algorithm in action from the starting point $(-4,\ -0.5)$.

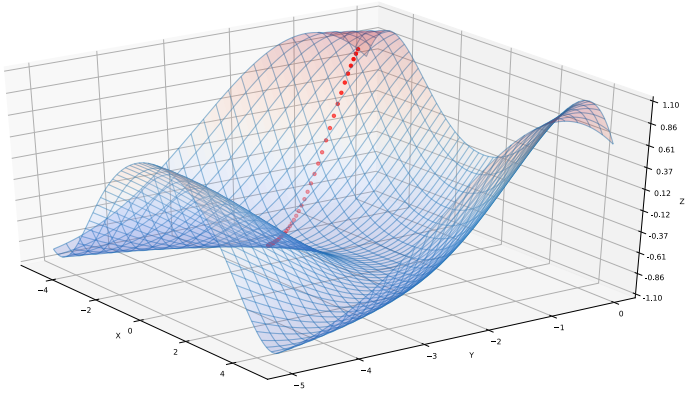


Figure 8: 3D gradient descent algorithm in action for $f(x,y) = \sin\left(0.1\ x^2 + 0.05\ y^2 + y\right)$

In fact, the gradient descent algorithm can be generalized in n-dimensional spaces. In n-dimensional spaces, the function f can be a function of n independent variables as:

$$f(x_0, x_1, \ldots, x_n) = f(\mathbf{x}) \quad \text{where} \quad \mathbf{x}(x_0, x_1, \ldots, x_n)$$

the gradient descent algorithm can be simply written in a vectorial form as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \nabla f(\mathbf{x}_k)$$

where $\nabla f(\mathbf{x})$ is the gradient vector of f defined as:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

To summarize, at this point we have an algorithm able to find the minimum of a continuum function in a n dimensional spaces. The gradient descent algorithm is extensively used in neural networks for minimizing loss functions. The next section will describe this process.

3.3 The loss function

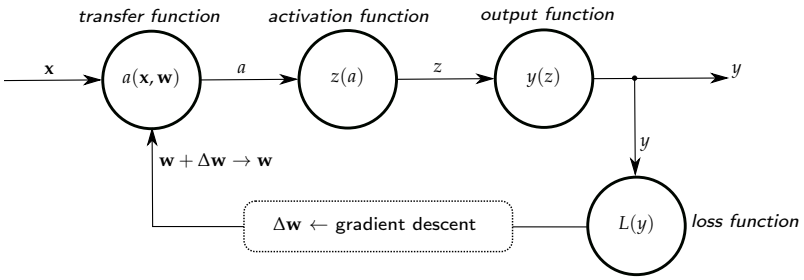
Loss functions are special functions able to measure the accordance of a prediction with their target values. High scores indicate bad predictions whereas low scores indicate good accordance of predictions. A common loss function (noted L) often used in machine learning is the Mean

Square Error (MSE). The MSE function L is defined as:

$$L(\mathbf{y}) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - t^{(i)}\right)^2$$

where m is the total number of data points, $y^{(i)}$ and $t^{(i)}$ are related to a unique instance of a prediction and its target value.

At this stage, it should be nice to remember the Perceptron architecture. This architecture is highlighted on the above scheme.



Here, the idea of the back propagation step is to minimize the loss function $L(\mathbf{y})$ in regard of weights $\mathbf{w}$. For sure, the gradient descent algorithm can be used to make this job. So, the gradient of the loss function in regard of weights have to be computed.

$$\nabla L(\mathbf{w}) = \left(\frac{\partial L}{\partial w_0}, \frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_n}\right)$$

If we focus on the first component of the gradient $\nabla L(\mathbf{w})$, it gives:

$$\frac{\partial L}{\partial w_0} = \frac{\partial}{\partial w_0}\left(\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - t^{(i)}\right)^2\right)$$

By the way, the derivative can be introduced in the sum as:

$$\frac{\partial L}{\partial w_0} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial w_0}\left(\left(y^{(i)} - t^{(i)}\right)^2\right)$$

Following the Perceptron architecture, one notes that the output y is a composed function of z and a as:

$$\begin{aligned} y &= y\,(\,z\,) \\ &= y\,(\,z(\,a\,)\,) \\ &= y\,(\,z\,(\,a(\mathbf{x},\mathbf{w})\,)\,) \end{aligned}$$

So, the following derivative chain rule can be written:

$$\frac{\partial f}{\partial w_0} = \frac{\partial f}{\partial y}\cdot\frac{\partial y}{\partial z}\cdot\frac{\partial z}{\partial a}\cdot\frac{\partial a}{\partial w_0} \quad \text{where} \quad f(y) = (y-t)^2$$

The derivative of each chain item can be easily computed:

$$\begin{aligned}
\frac{\partial f}{\partial y} &= \frac{\partial}{\partial y}\left((y-t)^2\right) && = 2(y-t) && \rightarrow \textit{ derivative of loss function (MSE)}\\
\frac{\partial y}{\partial z} &= \frac{\partial}{\partial z}(z) && = 1 && \rightarrow \textit{ derivative of output function (identity)}\\
\frac{\partial z}{\partial a} &= \frac{\partial}{\partial a}(\sigma(a)) = \sigma(a)(1-\sigma(a)) && = \sigma'(a) && \rightarrow \textit{ derivative of activation function (sigmoid)}\\
\frac{\partial a}{\partial w_0} &= \frac{\partial}{\partial w_0}(b + x_0 w_0 + x_1 w_1 + \cdots + x_n w_n) && = x_0 && \rightarrow \textit{ derivative of transfer function}
\end{aligned}$$

We can combine all these derivatives in one single formula. It gives:

$$\frac{\partial L}{\partial w_0} = \frac{1}{m}\sum_{i=1}^{m} 2(y^{(i)} - t^{(i)}) \cdot 1 \cdot \sigma'(a^{(i)}) \cdot x_0^{(i)}$$

This formula can be generalized to any weight w_j as:

$$\boxed{\frac{\partial L}{\partial w_j} = \frac{2}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)}) \cdot x_j^{(i)}}$$

and for the bias b:

$$\boxed{\frac{\partial L}{\partial b} = \frac{2}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)})}$$

ℹ *Here, the trick for computing quickly the derivative of L regarding the bias b is to consider b as a weight where its entry is always equal to 1 (see 5). Practically, in some machine learning frameworks, this feature is given by adding silently a new component to the* $\mathbf{x}$ *vector (which is always equal to a unit) and also to the weight* $\mathbf{w}$ *(which is equal to the bias). It gives:*

$$a = \begin{bmatrix} 1 \\ x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \cdot \begin{bmatrix} b \\ w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = b + x_0 w_0 + x_1 w_1 + \cdots + x_n w_n$$

So, it is equivalent to the classical transfer function. It allows to remove the bias for avoiding its special treatment. This trick is often used in the literature about neural network! For pedagogical reason this trick will not be used in this document.

⚠ *Don't be confuse, remember that n is related to the dimension of the entry vector* $\mathbf{x}(x_0, x_1, \cdots, x_n)$ *whereas m is related to the number of entry points. In other words, the learning step involves m points of n-dimensions.*

3.4 Implementation of the loss function and gradient descent algorithm

We stated, from the previous sections, that optimal values of weights and bias can be found thanks to the back propagation step that implements the gradient descent algorithm. This algo-

rithm gives:

$$
\begin{aligned}
w_0 &\rightarrow w_0 + \frac{2\eta}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)}) \cdot x_0^{(i)} \\
&\vdots \\
w_n &\rightarrow w_n + \frac{2\eta}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)}) \cdot x_n^{(i)} \\
b &\rightarrow b + \frac{2\eta}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)})
\end{aligned}
$$

For the weight part, the related formula can be expressed in a vectorial form as:

$$
\mathbf{w} \rightarrow \mathbf{w} + \frac{2\eta}{m}\sum_{i=1}^{m}(y^{(i)} - t^{(i)}) \cdot \sigma'(a^{(i)}) \cdot \mathbf{x}^{(i)}
$$

where $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are n-dimensional vectors. Now, as you can see in the above formula, we must implement a summation on all the m points. Based on this observation, the sum can be replaced thanks to *linear algebra* operation such as the *dot* products:

$$
\mathbf{w} \rightarrow \mathbf{w} + \frac{2\eta}{m}\left(\mathbf{X}^{\top} \cdot [(\mathbf{y} - \mathbf{t}) \circ \sigma(\mathbf{a}) \circ (1 - \sigma(\mathbf{a}))]\right)
$$

where:

- $\mathbf{w}$, $\mathbf{y}$, $\mathbf{t}$ and $\mathbf{a}$ are (n)-dimensional vectors,
- $\mathbf{X}$ is a $(n \times m)$-dimensional **matrix** (or tensor),
- $^{\top}$ is the transpose of a matrix (or a vector),
- $\circ$ is the *Hadamard* product (the so-called *element wise* product)
- $\cdot$ is the dot product (here, the dot product replaces the sum),
- n is the number of dimension of an entry $\mathbf{x}_i$ and
- m is the total number of entry $\mathbf{X}$.

> ℹ *Here, the advantage of using linear algebra (tensorial) operations such as Hadamard or dot products is to deal with large amount of data. Linear algebra computations involved in tensorial operations have been optimized for a long time and the main linear algebra algorithms can be massively parallelized in both CPU and GPU. Deep learning processes extensively use linear algebra computations. For example, the name of one of the must popular free deep learning framework (supported by Google) is "TensorFlow". For now,* ***be aware****, to follow the next part of this document, you must be familiar with tensorial operations and linear algebra computation rules.*

To clarify this last formula, we will take the first example of this document that implements the following data:

```
>>> x
array([[-2.69560892, -1.72976383],
       [-2.12983421, -1.78565134],
       [-0.71161969,  8.43909972],
       ...,
       [-1.31913952, -4.69944969],
       [-4.46749849, -4.19619198],
       [-1.83938485,  0.02932308]])
```

1,000 data points

x_0 x_1

```
>>> t
array([0,
       0,
       1,
       ...,
       0,
       0,
       0])
```

1,000 data labels'

t

Here, the number of dimension n of an entry $\mathbf{x}_i$ is 2 and the total number of entry $\mathbf{X}$ is 1,000. To summarize, in this example, $n = 2$ and $m = 1,000$. Now, let's focus on the trick able to replace a *sum* by a *dot product*. If we take the current example, we can highlights the main terms of the

previous equation:

$$\underbrace{\mathbf{w}}_{\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}} \rightarrow \underbrace{\mathbf{w}}_{\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}} + \overbrace{\frac{2\eta}{m}}^{\text{scalar}} \times \underbrace{\mathbf{X}^\top}_{\begin{bmatrix} x_{0_0} & x_{0_1} & \dots & x_{0_{999}} \\ x_{1_0} & x_{1_1} & \dots & x_{1_{999}} \end{bmatrix}} \cdot \underbrace{[(\mathbf{y}-\mathbf{t}) \circ \sigma(\mathbf{a}) \circ (1-\sigma(\mathbf{a}))]}_{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{999} \end{bmatrix}}$$

However, it does not correspond to the internal way of vector storage in Numpy. Numpy stores vectors as row vectors. So, the previous formula has to be rewritten as follows:

$$\underbrace{\mathbf{w}}_{\begin{bmatrix} w_0 & w_1 \end{bmatrix}} \rightarrow \underbrace{\mathbf{w}}_{\begin{bmatrix} w_0 & w_1 \end{bmatrix}} + \overbrace{\frac{2\eta}{m}}^{\text{scalar}} \times \underbrace{[(\mathbf{y}-\mathbf{t}) \circ \sigma(\mathbf{a}) \circ (1-\sigma(\mathbf{a}))]^\top}_{\begin{bmatrix} a_0 & a_1 & \dots & a_{999} \end{bmatrix}} \cdot \underbrace{\mathbf{X}}_{\begin{bmatrix} x_{0_0} & x_{0_1} \\ x_{1_0} & x_{1_1} \\ \vdots & \vdots \\ x_{999_0} & x_{999_1} \end{bmatrix}}$$

So, the gradient descent algorithm in Python can be written as:

$$\boxed{\mathbf{w} \rightarrow \mathbf{w} + \frac{2\eta}{m} \times [(\mathbf{y}-\mathbf{t}) \circ \sigma(\mathbf{a}) \circ (1-\sigma(\mathbf{a}))]^\top \cdot \mathbf{X}}$$

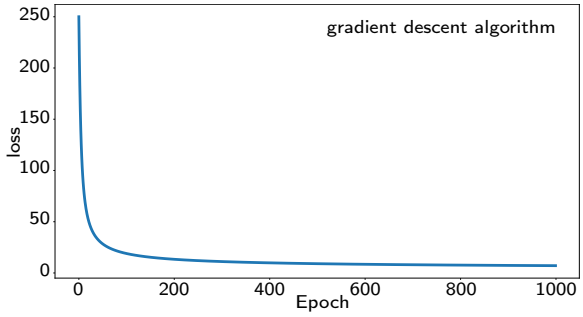
The next listing shows the implementation of the modified Perceptron algorithm with gradient descent and sigmoid activation function.

Listing 7: perceptron-sigmoid-gradient-descent.py

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    z = sigmoid(x)
    return z * (1 - z)

# make data
x,t = datasets.make_blobs(n_samples=1000, centers=2, n_features=2)

# initialize paramaters
w    = np.zeros(len(x[0])) # it contains (w0,w1)
b    = 0.
eta   = .1
epoch = 1000

# some useful variable
m    = len(t)
loss = np.zeros(epoch)

for i in range(epoch):
    # feed forward
    a = np.dot(x, w) + b
    z = sigmoid(a)
    y = z
```

```
    loss[i] = (1./m) * np.sum( (y-t)**2 ) # record the current value of loss

    # backward propagation
    w += -(2./m)*eta * np.dot(((y-t)*sigmoid_prime(a)).T, x)
    b += -(2./m)*eta * np.sum( (y-t)*sigmoid_prime(a))
```

You can note that the related code records the evolution of the loss function into the `loss` array. This is very important. It allows to check if the learning process is going right! The following graph shows an example of a "good" loss evolution.



As you can see the loss decreases continuously toward zero while the epoch number increases. It means that the back propagation algorithm is working as expected: it minimizes the loss by adjusting weights and bias values in a correct way thanks to the gradient descent algorithm.

3.5 Stochastic gradient descent algorithm with mini-batch

When the amount of data is very large, it may be difficult to treat all the data in one step. To avoid memory problem, the data could be sliced in k mini-batches. The following picture illustrates the mini-batch process. Here the data are sliced in two separated mini-batch.

```
>>> x
array([[-2.69560892, -1.72976383],
       [-2.12983421, -1.78565134],   } mini-batch 1
       [-0.71161969,  8.43909972],
       ...,
       [-1.31913952, -4.69944969],
       [-4.46749849, -4.19619198],   } mini-batch 2
       [-1.83938485,  0.02932308]])

>>> t
array([0,                            } mini-batch 1
       0,
       1,
       ...,
       0,                            } mini-batch 2
       0,
       0])
```

Another good reason for using mini-batches is for optimizing the gradient descent algorithm. As you know, gradient descent algorithm can be trapped in a local minima. Using mini-batch allows to introduce *noises* in the algorithm. It increases the chance to find the global minima instead of a local one. This process is called *stochastic gradient descent algorithm*. It is simple, elegant and really efficient! This strategy is massively used in machine learning. It consists in:

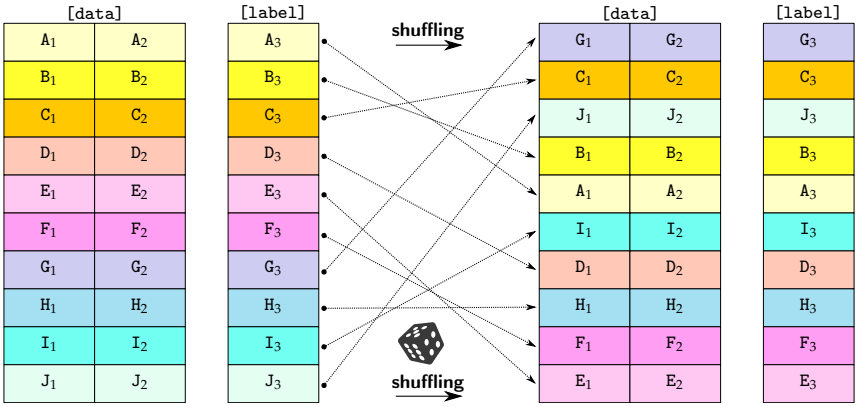
1. randomizing the data storage. It means that data arrays are shuffled.
2. slicing the newly organized data and label.

3.5.1 Shuffling data with numpy

Since the beginning of this document, we have seen that data are stored in two arrays:

1. the `data` array that contains the entry point and
2. the `label` array that contains the target values.

The shuffling operation consists in reordering randomly these arrays. Indeed, special attention must be taken. The shuffling operation must keep the correspondence between the indices of the `data` and the `label` arrays. The next figure summarizes this process.



Again, Numpy gives all we need! Let's make a sample code into the interpreter:

```
>>> data = np.arange(0,100,10)
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
>>> label = np.arange(0,1000,100)
array([  0, 100, 200, 300, 400, 500, 600, 700, 800, 900])
```

Now, we create a new array that contains the index of the `data` and `label` arrays:

```
>>> s = np.arange(data.shape[0])
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Let's shuffling this array thanks to the `np.random.shuffle` function:

```
>>> np.random.shuffle(s)
>>> s
array([7, 0, 9, 5, 8, 4, 2, 6, 1, 3])
```

Now, we can use this shuffled arrays as indices for re-organizing the `data` and `label` arrays as follow:

```
>>> data[s]
array([70 , 0, 90 , 50 , 80 , 40 , 20 , 60 , 10 , 30])
>>> label[s]
array([700, 0, 900, 500, 800, 400, 200, 600, 100, 300])
```

As you can see both `data` and `label` has been reorganized with the same indices. It is very important, because we need to keep the correspondence between the items contained in `data` and `label`. Now, we want to slice these new arrays in five mini-batches. We can use the `np.split` function as:

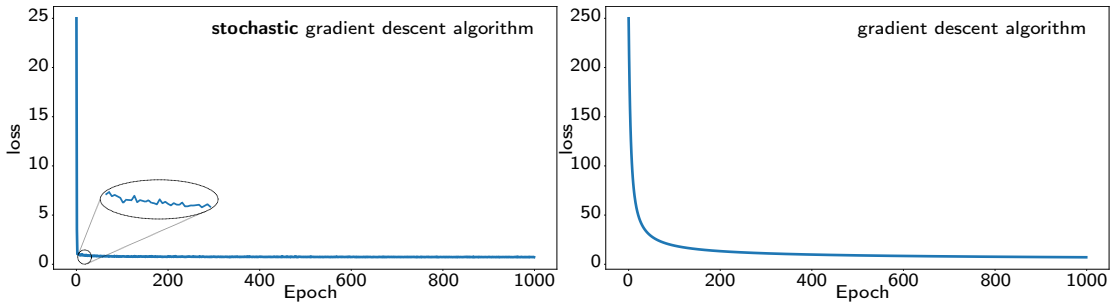
```
>>> mb = 5 # number of mini-batch
>>> data_s = np.split(data[s], mb)
[array([70,  0]), array([90, 50])  , array([80, 40])  , array([20, 60])  , array([10, 30])]
>>> label_s = np.split(label[s], mb)
[array([700, 0]), array([900, 500]), array([800, 400]), array([200, 600]), array([100, 300])]
```

Here, we have built five mini-batches with shuffled data. Now, we can introduce these shuffled data into the modified Perceptron algorithm. The next listing shows the implementation of the stochastic gradient descent algorithm into a modified Perceptron algorithm.

Listing 8: perceptron-sigmoid-gradient-descent-mini-batch.py

```
# initialize paramaters
w     = np.zeros(len(x[0])) # it contains (w0,w1)
b     = 0.
eta   = .1
epoch = 1000
mb    = 10 # number of mini-batch

# some useful variable
loss = np.zeros(epoch)
m    = len(t)

for i in range(epoch):
    s = np.arange(x.shape[0])
    np.random.shuffle(s)
    x_s = np.split(x[s], mb)
    t_s = np.split(t[s], mb)
    l = 0.
    for xb,tb in zip(x_s, t_s):
        m = len(t_s)

        # feed forward
        a = np.dot(xb, w) + b
        z = sigmoid(a)
        y = z

        l += (1./m) * np.sum( (y-tb)**2 ) # record the current value of loss

        # backward propagation
        w -= (2./m)*eta * np.dot(((y-tb)*sigmoid_prime(a)).T, xb)
        b -= (2./m)*eta * np.sum( (y-tb)*sigmoid_prime(a))
    loss[i] = l
```

The related algorithm gives the following evolution of loss (left image).



As you can see, the evolution is a little bit noisy but the stochastic gradient descent method is quicker than the non-stochastic one.

4 A note on pre-processing data

Preparing the data is really important for machine learning. So, please remember this simple law: **bad data = bad predictions**. In real life, this process is the most time-consuming step of machine learning. You need to automatize the process of collecting data, removing outlier points, removing non-useful data and so on... It can be really complicated!

In this document, we deal only with already-prepared data available in machine learning frameworks. By consequent, this preliminary work, that consists in cleaning data, was already done

for you. In this document, we will not focus on this particular job. However, you have to keep in mind that this is a very important work that must be done on real data set.

However, even if your data is clean, you can apply some numerical treatments in order to optimize the learning process. This step is called *feature scaling*. It consists in scaling data in a reasonable range close to $[0,1]$. This process allows to facilitate the learning step.

Several method exists. To get an overview, you can visit the *wikipedia page* [5] that describes several methods. My favorite choice came from philosophical reason: in science, it is considered that *nature* often gives us data that follows normal distributions. In such cases, a good *feature scaling* is the *standardization* defined as:

$$x \to \frac{x - \bar{x}}{\sigma} \tag{11}$$

where x are the original data, $\bar{x}$ the mean data value and σ the standard deviation. Thanks to numpy, *standardization* is really easy to obtain:

```
x = (x - x.mean()) / x.std()
```

where x is a numpy array that contains data set.

4.1 The complete code of the modified Perceptron algorithm

At this time, we have learned a lot of things. Based on the original Perceptron algorithm which is the most simple neural network, we have seen how the learning process thanks to the feed forward and back propagation step. This architecture is very general and should be applied to a wide range of machine learning strategies. After that, thanks to differentiable activation functions such as sigmoid function, we have seen an optimal back propagation strategy that implements loss function and stochastic gradient descent algorithm with mini-batches. Finally, we have discussed shortly about data preparation. All these ingredients enable the implementation of a complete mono-neural network Perceptron as shown on the next listing.

Listing 9: perceptron-complete.py

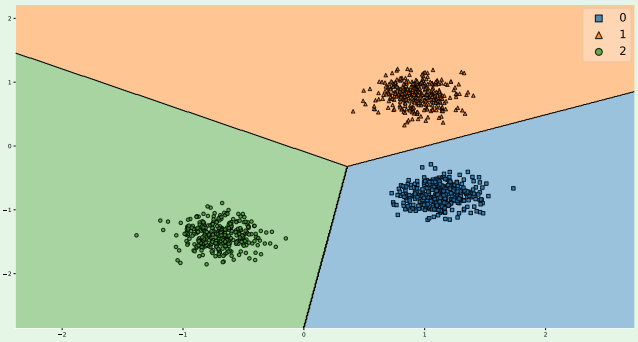
```
import numpy as np
from sklearn import datasets
from matplotlib import pyplot as plt
from mlxtend.plotting import plot_decision_regions

# if you want to get previous data
x  = np.fromfile('perceptron-x.dat')
y  = np.fromfile('perceptron-y.dat')
t = np.fromfile('perceptron-label.dat', dtype=int)
x = np.stack((x, y), axis=-1)


def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    z = sigmoid(x)
    return z * (1 - z)

# make data
x,t = datasets.make_blobs(n_samples=1000, centers=2, n_features=2)

# standardize the data (a method to get more robust data)
x = (x - x.mean()) / x.std()
```

[5] https://en.wikipedia.org/wiki/Feature_scaling

```
# initialize paramaters
w     = np.random.rand(len(x[0])) # it is better to get random values here
b     = 0.
eta   = .1
epoch = 1000
mb    = 10 # number of mini-batch

# some useful variable
loss = np.zeros(epoch)
m    = len(t)

for i in range(epoch):
    s = np.arange(x.shape[0])
    np.random.shuffle(s)
    x_s = np.split(x[s], mb)
    t_s = np.split(t[s], mb)
    l = 0.
    for xb,tb in zip(x_s, t_s):
        m = len(t_s)

        # feed forward
        a = np.dot(xb, w) + b
        z = sigmoid(a)
        y = z

        l += (1./m) * np.sum( (y-tb)**2 ) # record the current value of loss

        # backward propagation
        w -= (2./m)*eta * np.dot(((y-tb)*sigmoid_prime(a)).T, xb)
        b -= (2./m)*eta * np.sum( (y-tb)*sigmoid_prime(a))
    loss[i] = l

# the perceptron class, this is mandatory for plotting with mlxtend
class Perceptron:
    @staticmethod
    def predict(X): # here, X is a numpy array that contains all the data
        a = np.dot(X, w) + b
        z = sigmoid(a)
        z = np.where(z >= 0.8, 1, (np.where(z<=0.2, 0, -1)))
        return z

# plotting
plt.figure()
plot_decision_regions(x, t, clf=Perceptron)

plt.figure()
plt.plot(loss, '-')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

As you can see, a single neural network is not very hard to implement, therefore it uses a lot of tips and tricks. A last trick used in this listing is to initialize the weights to random values in the range $[0,1]$. It allows to optimize the stochastic gradient descent algorithm **for already scaled data**.

☞ *Now, as an exercise, you can try to implement the proposed Perceptron code (see previous listing) into a Python class (see section 4.3 about object oriented programming). After that, you can use this class to solve linear multi-classification problems with several instances of Perceptron (one for each class). For three classes problem you must obtain something like this:*



Correction is available here : `perceptron-multi-class.py`

The further sections of this document will be dedicated to multiple labels (more than two classes) classification and non linearly separable data using multi-layer neural networks.

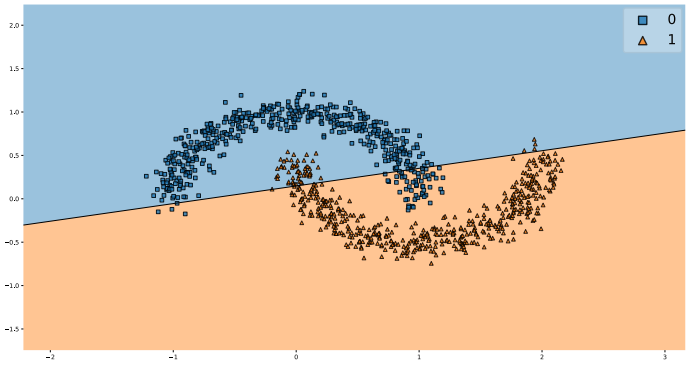
5 Multi-layer neural networks

At this step, we are able to classify linearly separable data in multiple classes. Indeed, in many cases, for real complex problems, the related data are not fully linearly separable. In such problems, multi-layer neural networks can be used.

To illustrate non-linearly separable data problems, we will use here another data set generator. Instead of the `datasets.make_blobs` function we will use the `datasets.make_moons` function.

☞ *To test it by yourself, replace the* `datasets.make_blobs` *function by the* `datasets.make_moons` *inside the initial Perceptron code :* `perceptron-init.py`.

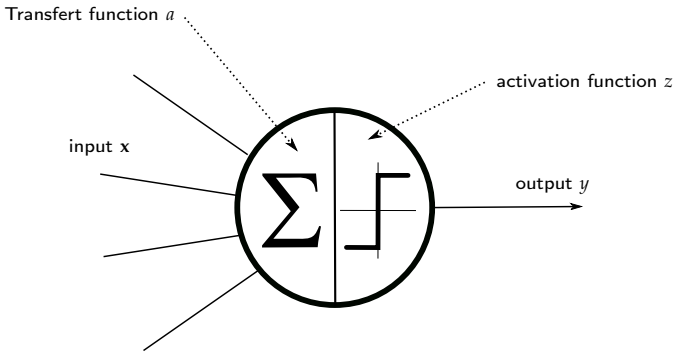
As a result, it should give the following chart:



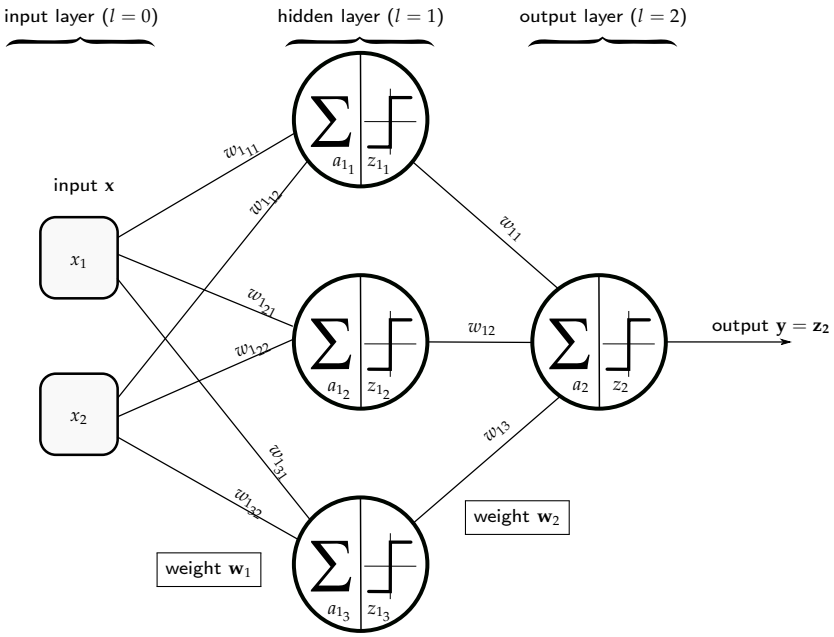
As you can see, this data set is not linearly separable. So, our Perceptron algorithm is not able to classify correctly this kind of data. To solve this problem, we should implement multi-layer dense neural networks. This kind of networks is able to deal with non-linear problems.

In fact, our well-known Perceptron algorithm can be seen as a fundamental brick for building

multi-layer neural networks able to deal with complex problems. Here, we are going to build a two layers dense neural network to solve this particular problem. Now, we will draw a neuron as follow:



Here, a neuron is drawn as a circle that contains the transfer function $a(x)$ and the activation function $z(a)$. A neuron accepts multiple inputs and gives one output. To solve our non-linear classification problem with the *moon* data set, we will implement the following architecture:



- The input layer **x** is connected to three neurons. This neuron layer is called *hidden* layer.
- The *hidden* layer neurons are connected to one neuron called *output* layer.

This network forms a *dense multi-layer* neural network able to deal with non-linear separable data set. A special attention must be given to the number of related weights and biases. In a general manner, imagine a layer of rank $l-1$ densely connected to its next layer of rank l. Now, suppose that the number of neurons that compose the layer of rank l is noted N_l. By the way, the total number of weight $N_{(l-1)\to(l)}$ that connect the layer of rank $(l-1)$ to its next one (l) is:

$$N_{(l-1)\to(l)} = N_{l-1} \times N_l$$

If we apply this simple formula to our architecture, the numbers of weight for each connection

are:

$$N_{0\to1} = N_0 \times N_1 = 2 \times 3 = 6$$
$$N_{1\to2} = N_1 \times N_2 = 3 \times 1 = 3$$

Instead of storing the weights as vector, a more practical data storage involves matrices. So, these weights can be written as matrices where :

- the length of the first axis is the number of neuron N_{l-1} of its entry layer and
- the length of the second axis is the number of neuron N_l of the output layer.

In the above case, it gives:

$$\mathbf{W}_1 = \begin{bmatrix} w_{1_{11}} & w_{1_{21}} & w_{1_{31}} \\ w_{1_{12}} & w_{1_{22}} & w_{1_{32}} \end{bmatrix} \quad \text{and} \quad \mathbf{W}_2 = \begin{bmatrix} w_{2_{11}} \\ w_{2_{12}} \\ w_{2_{13}} \end{bmatrix}$$

> *Note on mathematical conventions:*
> - *scalars are written as italic-lowercase typeface, e.g,* w
> - *vectors are written as bold-lowercase typeface, e.g,* $\mathbf{w}$
> - *matrices or tensors are written as italic-uppercase typeface, e.g,* $\mathbf{W}$

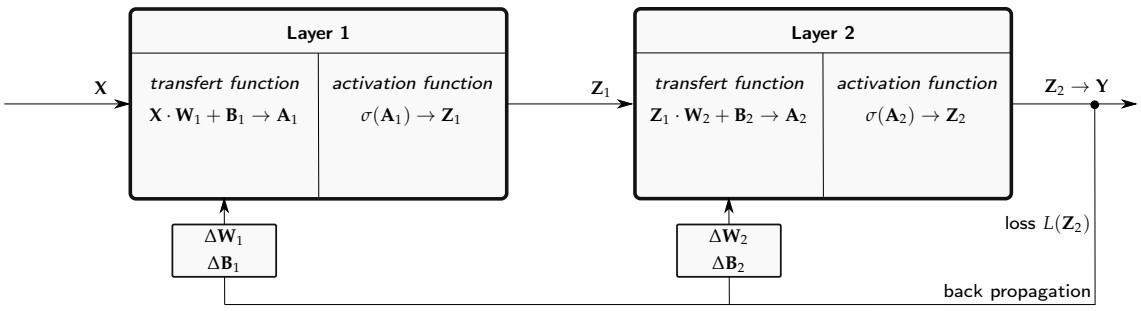
In general manner, weight matrices $\mathbf{W}_l$ (from layer $l-1$ to layer l) are written as:

$$\overbrace{\begin{Bmatrix} \\ l \text{ rows = entry number } N_{l-1} \\ \\ \end{Bmatrix}}^{k \text{ col = output number } N_l} \begin{bmatrix} w_{l_{11}} & w_{l_{21}} & \cdots & \cdots & w_{l_{k1}} \\ w_{l_{12}} & w_{l_{22}} & \cdots & \cdots & w_{l_{k2}} \\ \vdots & \vdots & \cdots & \cdots & \vdots \\ w_{l_{1j}} & w_{l_{2j}} & \cdots & \cdots & w_{l_{kl}} \end{bmatrix} = \mathbf{W}_l$$

Now, we will show how to implement the feed forward step with matrices.

5.1 Initialization and implementation of the feed forward step

We will move on higher level of abstraction. A more convenient and general point of view is adopted by using matrices instead of vectors or scalars. Each layer is considered as a box where inputs and outputs are highlighted by arrows. With this new visualization, the previous neural network scheme becomes:



Here, single neurons are hidden; all the symbols are considered as matrices instead of scalars and vectors. Let's detail the matrix shapes associated to each symbols of the above diagram. For

the current example it gives the following shapes.

- $\mathbf{X}$ is a $(2 \times m)$ matrix because it contains the m points of the mini-batch as follows:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix} \tag{12}$$

- $\mathbf{A_1}$ and $\mathbf{A_2}$ are respectively $(m \times 3)$ and $(m \times 1)$ matrices as follows:

$$\mathbf{A}_1 = \begin{bmatrix} a_{1_1}^{(1)} & a_{1_2}^{(1)} & a_{1_3}^{(1)} \\ \vdots & \vdots & \vdots \\ a_{1_1}^{(m)} & a_{1_2}^{(m)} & a_{1_3}^{(m)} \end{bmatrix} \quad \text{and} \quad \mathbf{A}_2 = \begin{bmatrix} a_2^{(1)} \\ \vdots \\ a_2^{(m)} \end{bmatrix}$$

- $\mathbf{Z_1}$ and $\mathbf{Z_2}$ are respectively $(m \times 3)$ and $(m \times 1)$ matrices as follows:

$$\mathbf{Z}_1 = \begin{bmatrix} z_{1_1}^{(1)} & z_{1_2}^{(1)} & z_{1_3}^{(1)} \\ \vdots & \vdots & \vdots \\ z_{1_1}^{(m)} & z_{1_2}^{(m)} & z_{1_3}^{(m)} \end{bmatrix} \quad \text{and} \quad \mathbf{Z}_2 = \begin{bmatrix} z_2^{(1)} \\ \vdots \\ z_2^{(m)} \end{bmatrix}$$

- $\mathbf{W_1}$ and $\mathbf{W_2}$ are respectively (2×3) and (3×1) matrices as follows:

$$\mathbf{W}_1 = \begin{bmatrix} w_{1_{11}} & w_{1_{21}} & w_{1_{31}} \\ w_{1_{12}} & w_{1_{22}} & w_{1_{32}} \end{bmatrix} \quad \text{and} \quad \mathbf{W}_2 = \begin{bmatrix} w_{2_{11}} \\ w_{2_{12}} \\ w_{2_{13}} \end{bmatrix}$$

- $\mathbf{B_1}$ and $\mathbf{B_2}$ are respectively (1×3) and (1×1) matrices as follows:

$$\mathbf{B}_1 = \begin{bmatrix} b_{1_1} & b_{1_2} & b_{1_3} \end{bmatrix} \quad \text{and} \quad \mathbf{B}_2 = \begin{bmatrix} b_2 \end{bmatrix}$$

Let's go back to the code. Following the previous comments, the weights can be initialized as matrices:

```
W1 = np.random.random( (2, 3) )
W2 = np.random.random( (3, 1) )
```

The length of the first axis corresponds to the number of entry and the length of the second axis corresponds to the number of output of the considered layer. Bias can be initialized as:

```
B1 = np.zeros( (1, 3) )
B2 = np.zeros( (1, 1) )
```

You can note here that the first axis length of bias is always equal to one because there is only one bias by neuron. Now, implementing the feed forward step is quite simple. In our case, it gives:

```
        # feed forward (hidden layer)
        A1 = np.dot(X, W1) + B1
        Z1 = sigmoid(A1)
        # feed forward (output layer)
        A2 = np.dot(Z1, W2) + B2
        Z2 = sigmoid(A2)
```

5.2 The backpropagation step

The most difficult part of multi-layer neural network is the implementation of the back propagation step through the gradient descent algorithm. We take here a modified MSE loss function that gives:

$$L(\mathbf{Z}_2) = \frac{1}{2}\sum_{i=1}^{m}\left(z_2^{(i)} - t^{(i)}\right)^2$$

In the above case, the gradient of the loss function regarding $\mathbf{W}_1$ and $\mathbf{W}_2$ give the following tensors:

$$\begin{aligned}\nabla L(\mathbf{W}_1) &= \begin{bmatrix} \frac{\partial L}{\partial w_{1_{11}}} & \frac{\partial L}{\partial w_{1_{21}}} & \frac{\partial L}{\partial w_{1_{31}}} \\ \frac{\partial L}{\partial w_{1_{12}}} & \frac{\partial L}{\partial w_{1_{22}}} & \frac{\partial L}{\partial w_{1_{32}}} \end{bmatrix} \\ \nabla L(\mathbf{W}_2) &= \begin{bmatrix} \frac{\partial L}{\partial w_{2_{11}}} & \frac{\partial L}{\partial w_{2_{12}}} & \frac{\partial L}{\partial w_{2_{13}}} \end{bmatrix}\end{aligned} \tag{13}$$

5.2.1 Backpropagation of the second weight layer $\mathbf{W}_2$

Now, we focus on the derivative of the loss regarding $\mathbf{W}_2$. For the first component of $\nabla L(\mathbf{W}_2)$, it gives:

$$\frac{\partial L}{\partial w_{2_{11}}} = \sum_{i=1}^{m}\frac{\partial}{\partial w_{2_{11}}}\left(\frac{1}{2}\left(z_2^{(i)} - t^{(i)}\right)^2\right) \tag{14}$$

So, the following derivative chain rule can be written:

$$\frac{\partial f}{\partial w_{2_{11}}} = \frac{\partial f}{\partial z_2}\cdot\frac{\partial z_2}{\partial a_2}\cdot\frac{\partial a_2}{\partial w_{2_{11}}} \quad \text{where} \quad f(z_2) = \frac{1}{2}(z_2 - t)^2$$

It gives:

$$\frac{\partial f}{\partial w_{2_{11}}} = (z_2 - t)\cdot\sigma'(a_2)\cdot z_{1_1}$$

So, when retrieving the sum, it gives:

$$\frac{\partial L}{\partial w_{2_{11}}} = \sum_{i=1}^{m}\left(z_2^{(i)} - t^{(i)}\right)\cdot\sigma'(a_2^{(i)})\cdot z_{1_1}^{(i)}$$

Finally, the gradient descent algorithm could be written in a tensorial form as:

$$\mathbf{W}_2 \rightarrow \mathbf{W}_2 - \eta\left[\mathbf{Z}_1^{\intercal}\cdot\left(\underbrace{(\mathbf{Z}_2 - \mathbf{T})}_{\texttt{L2}}\circ\underbrace{\sigma'(\mathbf{A}_2)}_{\texttt{M2}}\right)\right]$$

and the related Python code is:

```
## backward propagation of output layer
L2  = Z2-T
M2  = sigmoid_prime(A2)
## Compute gradients
Dw2 = np.dot(Z1.T, L2*M2)
Db2 = np.sum(L2*M2, axis=0)
## Change weights and bias
w2 -= eta * Dw2
```

```
        b2 -= eta * Db2
```

5.2.2 Backpropagation of the hidden weight layer $\mathbf{W}_1$

In a similar way, the derivative chain can be applied to $\mathbf{W}_1$ as:

$$\frac{\partial f}{\partial w_{1_{11}}} = \frac{\partial f}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_{1_1}} \cdot \frac{\partial z_{1_1}}{\partial a_{1_1}} \cdot \frac{\partial a_{1_1}}{\partial w_{1_{11}}} \quad \text{where} \quad f(z_2) = \frac{1}{2}(z_2 - t)^2$$

Be aware, in the above formula, as $w_{1_{11}}$ is an input of the first neuron of the hidden layer, the chain rule must contain the related activation function z_{1_1} and a_{1_1}. So, the above formula gives:

$$\frac{\partial f}{\partial w_{1_{11}}} = (z_2 - t) \cdot \sigma'(a_2) \cdot w_{2_{11}} \cdot \sigma'(a_{1_1}) \cdot x_1$$

Again, this derivative chain rule must be included in the sum as:

$$\frac{\partial L}{\partial w_{1_{11}}} = \sum_{i=1}^{m} \left(z_2^{(i)} - t^{(i)}\right) \cdot \sigma'(a_2^{(i)}) \cdot \cdot w_{2_{11}} \cdot \sigma'(a_{1_1}^{(i)}) \cdot x_1^{(i)}$$

Finally, the gradient descent algorithm could be written in a tensorial form as:

$$\mathbf{W}_1 \rightarrow \mathbf{W}_1 - \eta \left[\mathbf{X}^\intercal \cdot \left(\overbrace{\left((\mathtt{L2} \circ \mathtt{M2}) \cdot \mathbf{W}_2^\intercal\right)}^{\mathtt{L1}} \circ \overbrace{\sigma'(\mathbf{A}_1)}^{\mathtt{M1}} \right) \right]$$

and the related Python code is:

```
        ## backward propagation of hidden layer
        L1 = np.dot(L2*M2, w2.T)
        M1 = sigmoid_prime(A1)
        ## Compute gradients
        Dw1 = np.dot(X.T, L1*M1)
        Db1 = np.sum(L1*M1, axis=0)
        ## Change weights and bias
        w1 -= eta * Dw1
        b1 -= eta * Db1
```

5.3 Full implementation code

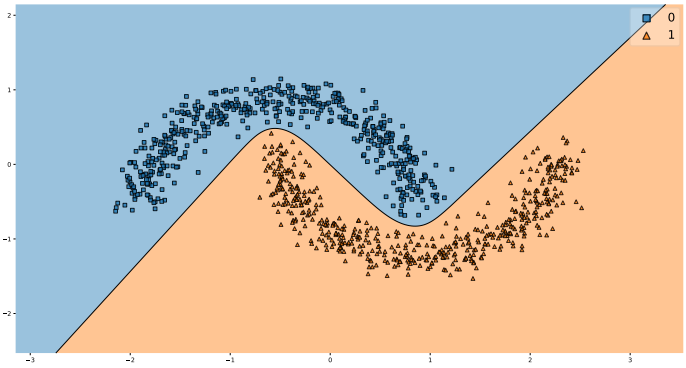
The full implementation of the multi-layer neural network able to deal with non-linear separable data is given in the next listing.

Listing 10: `multi-layer.py`

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    z = sigmoid(x)
    return z * (1 - z)

# make data
x,label = datasets.make_moons(n_samples=1000, noise=0.1)
```

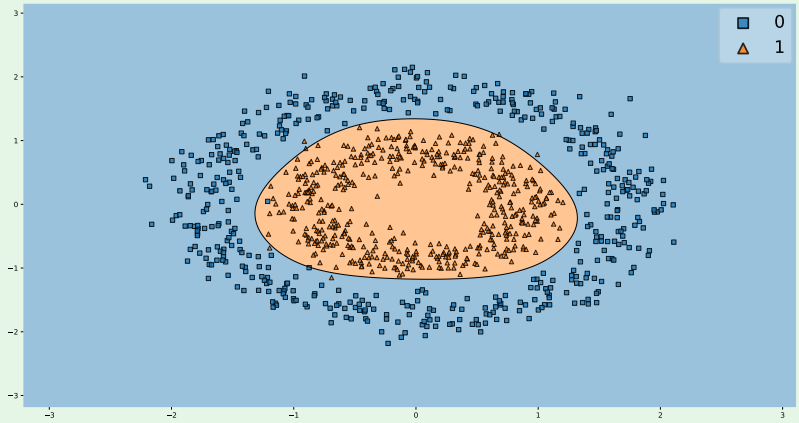
```
m = len(label)
x = (x - x.mean()) / x.std()

t = np.array([label])
t = t.astype(float)
t = t.T


# initialize weight
W1 = np.random.random((len(x[0]), 3))
W2 = np.random.random((3, 1))

#initialize bias
B1 = np.zeros((1, 3))
B2 = np.zeros((1, 1))

eta   = .1
epoch = 1000
mb    = 10 # number of mini-batch

# some useful variable
loss = np.zeros(epoch)

for i in range(epoch):
    s = np.arange(x.shape[0])
    np.random.shuffle(s)
    x_s = np.split(x[s], mb)
    t_s = np.split(t[s], mb)
    l = 0.
    for X,T in zip(x_s, t_s):
        m = len(t_s)

        # feed forward
        A1 = np.dot(X, W1) + B1
        Z1 = sigmoid(A1)

        A2 = np.dot(Z1, W2) + B2
        Z2 = sigmoid(A2)

        y = Z2
        l += (1./m) * np.sum( (y-T)**2 ) # record the current value of loss

        ## backward propagation of output layer
        L2  = Z2-T
        M2  = sigmoid_prime(A2)
        ## Compute gradients
        DW2 = np.dot(Z1.T, L2*M2)
        DB2 = np.sum(L2*M2, axis=0)
        ## Change weights and bias
        W2 -= eta * DW2
        B2 -= eta * DB2

        ## backward propagation of hidden layer
        L1 = np.dot(L2*M2, W2.T)
        M1 = sigmoid_prime(A1)
        ## Compute gradients
        DW1 = np.dot(X.T, L1*M1)
        DB1 = np.sum(L1*M1, axis=0)
        ## Change weights and bias
        W1 -= eta * DW1
        B1 -= eta * DB1
        #code.interact(local=locals())

    loss[i] = l
```

The full python code is also downloadable at [6]. As you can see in the following chart, the provided algorithm is able to classify non-linear separable data provided by the `make_moons` function.



☞ *To train yourself, you can try to add the following features to the previous code:*

1. *monitoring of the loss versus epoch*
2. *change the* `make_moons` *generator by* `make_circle`*. You must obtain something like this:*



Now, we will move on a more practical and real case. This is a multi-classification problem that involves hand-written digits.

6 Hello world of ML: multi-classification of hand-written digits

This section will be dedicated to a practical work. This is a direct application of what we have learned in the previous sections! This problem is inspired from a real one and is considered as the `"hello world"` of machine learning. It consists in making a machine able to recognize hand-written digits.

6.1 Autopsy of data

Here, we will use already-classified data that comes from the `datasets.load_digits` function of the `sklearn` module. Let's take a look on this data with the interpreter:

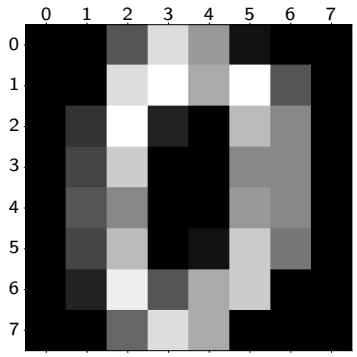
[6] `multi-layer.py`

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
>>> print(digits.data.shape)
(1797, 64)
```

As you can see in the above code, the `data` variable is a numpy array that contains 1797 values of 64-dimensional data. For pedagogical reasons, we have still treated only 2-dimensional data compatible with (X,Y) 2D plotting. Here, it is not possible to draw all this data on a single chart. However, we can plot the first data set as follows:

```
>>> import matplotlib.pyplot as plt
>>> plt.gray()
>>> plt.matshow(digits.images[0])
>>> plt.show()
```

You must see something like this:



As you can see, this first data set is related to a hand written digit '0'. Let's see how this image is stored.

```
>>> print(digits.images[0].shape)
(8, 8)
```

The handwritten digit was digitized in black and white using (8×8) matrices. As you can see in the previous figure, the first component of the matrix of coordinates $(0,\ 0)$ corresponds to the pixel located at the top-left corner of the image. Let's see the content of this matrix.

```
>>> print(digits.images[0])
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

As you can see, each pixel of the image takes an integer value in the range $[0, 1, 2, \ldots, 15]$. These values are related to the pixel grayscale. It means that 16 different gray values are allowed. The following grayscale drawing highlights the different grays with their related values. It starts from 0, that corresponds to the full black color and it ends by 15 that corresponds to a full white color.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

However, it is not possible to train a neural network with this kind of data. Remember that the input of a neural network for one single data set i is not a matrix but a vector as:

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & \dots & x_n^{(i)} \end{bmatrix} \tag{15}$$

So, image matrices must be flattened from (8×8) matrices to $(8 \times 8) = (64)$ length vectors. This kind of data was already formatted for us in the `digits.data` variable. As you can see in the following listing, the `digits.data` contains the previous matrix in a flattened format.

```
>>> print(digits.data[0])
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.
  0.  0.  3. 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.
  8.  0.  0.  5.  8.  0.  0.  9.  8.  0.  0.  4. 11.  0.  1.
 12.  7.  0.  0.  2. 14.  5. 10. 12.  0.  0.  0.  0.  6. 13.
 10.  0.  0.  0.]
```

> ℹ *Note that this process can be easily given thanks to the `flatten()` method of the `numpy.array` class. The following snippet shows its usage.*
>
> ```
> >>> print(digits.images[0].flatten())
> [ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.
>   0.  0.  3. 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.
>   8.  0.  0.  5.  8.  0.  0.  9.  8.  0.  0.  4. 11.  0.  1.
>  12.  7.  0.  0.  2. 14.  5. 10. 12.  0.  0.  0.  0.  6. 13.
>  10.  0.  0.  0.]
> ```

To conclude this part, the `digits.data` numpy array contains 1797 grayscale images. Each image is stored in a 64 length vector that corresponds to the flattened format of a (8×8) matrix of gray-scale pixels. The grayscale is coded with 16 different integer numbers in the $[0, 1, 2, \dots, 15]$ range.

> ℹ *In fact, the gray scale is coded on 4 bits* $\rightarrow 2^4 = 16$*!*

6.2 Autopsy of the related labels

Now, we will focus on data classification. The classification is accessible from the `digits.target` arrays. As expected, the `digits.target` contains 1797 items.

```
>>> print(digits.target.shape)
(1797,)
```

Each item contains a number in the $[0, 1, 2, \dots, 9]$ range that corresponds to the hand-written digit. Note that the classification was done by humans.

> ℹ *Machine learning processes that involve already-classified data by humans are known as* ***supervised*** *learning.*

```
>>> print(digits.target)
[0 1 2 ..., 8 9 8]
```

For example, if we take the first item number:

```
>>> print(digits.target[0])
0
```

as expected, you can see that it corresponds to the zero value which is the value of the handwritten digit (see the image in the previous section). However, based on our previous implementations, neural networks can deal only with binary classification problem which the expected answer is YES or NO.

6.3 from multi-classification to binary classification problem

As shown in the previous section, the proposed format of label classification does not match our expectations. Here, we are facing a multi-classification problem. An input must be classified into several classes. Here, ten classes are possible. They correspond to the digit numbers $[0, 1, 2, \ldots, 9]$. To fix this issue, a possible trick is to transform this multi-classification problem into 10 different binary classification problems.

In other words, we will transform the initial question (that can take 10 different answers):

- *which digit number is written on this image?* ..... 0,1,2,3,4,5,6,7,8 or 9?

into 10 different questions which required binary answers:

- *is zero written on this image?* ..... Yes or No?
- *is one written on this image?* ..... Yes or No?
- *is two written on this image?* ..... Yes or No?
- *is three written on this image?* ..... Yes or No?
- *is four written on this image?* ..... Yes or No?
- *is five written on this image?* ..... Yes or No?
- *is six written on this image?* ..... Yes or No?
- *is seven written on this image?* ..... Yes or No?
- *is eight written on this image?* ..... Yes or No?
- *is nine written on this image?* ..... Yes or No?

To make it possible, the `digits.target` variable must be changed. Here, each single value of the `digits.target` array that can take the `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'` or `'9'` values is replaced by a ten length array that can take `'0'` or `'1'`. This process can be summarized as:

$$\underbrace{5}_{\text{single value: 5}} \rightarrow \underbrace{[0,\ 0,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0]}_{\text{10 length array where the 5}^{\text{th}}\text{ item is 1}}$$

This transformation must be done to all values stored in the `digits.target` array. It can be easily done thanks to `np.eye()` function that comes from Numpy. It gives:

```
>>> print(digits.target)
[0 1 2 ..., 8 9 8]
>>> label = np.eye(10)[digits.target]
>>> print(label)
[[ 1.  0.  0. ...,  0.  0.  0.]
 [ 0.  1.  0. ...,  0.  0.  0.]
 [ 0.  0.  1. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  1.  0.]
 [ 0.  0.  0. ...,  0.  0.  1.]
 [ 0.  0.  0. ...,  0.  1.  0.]]
```

It's really a nice trick! Again, Numpy is really incredible :)

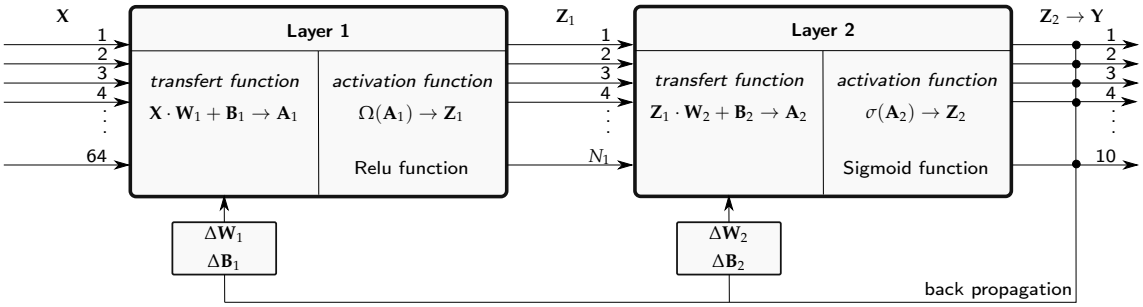
6.4 Try it yourself!

Let's suppose the following architecture composed by a two levels neural network, where:

- the number of entry $\mathbf{X}$ is 64,
- the number (noted N_1) of hidden neurons $\mathbf{Z}_1$ is unknown,
- the number of output $\mathbf{Y}$ is 10,
- the first activation function is the *Relu* function defined as:

$$\Omega(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

- the second activation function is the well-known *Sigmoid* function.



Now, based on our previous implementation, you can code this neural network by yourself . To make it, you can follow the proposed steps.

task1. Similarly to the `sigmoid()` and `sigmoid_prime()` functions, implement the `relu()` and `relu_prime()` functions. You should obtain something like this:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    z = sigmoid(x)
    return z * (1 - z)

def relu(x):
    ## ... YOUR IMPLEMENTATION HERE ...

def relu_prime(x):
    ## ... YOUR IMPLEMENTATION HERE ...
```

task2. Get the data thanks to the `load_digits()` function that comes from the `sklearn.datasets` module. Place the input data in a variable named `x` and the labels in a variable named `t`. You should obtain something like this:

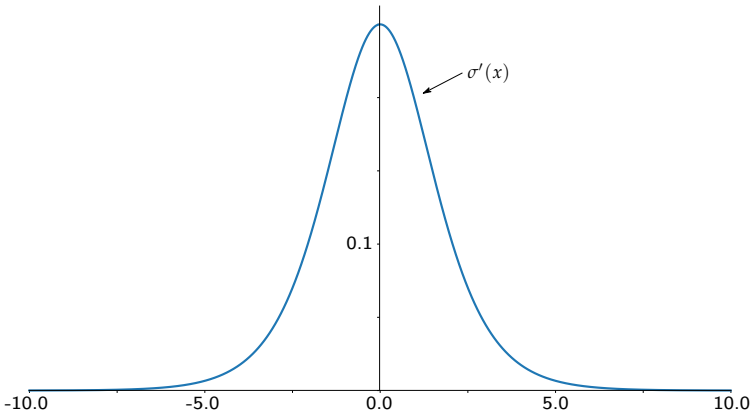
```
digits = datasets.load_digits()
x =  ## ... YOUR IMPLEMENTATION HERE ...
     ## ...
t =  ## ... YOUR IMPLEMENTATION HERE ...
     ## ...
```

Don't forget to apply the treatments described in the last section!

task3. Initialize weight and bias. In a first try, you can take the number of hidden neurons $N_1 = 10$. You should obtain something like this:

```
# number of outputs
N1 = # ... YOUR NUMBER HERE ... # <- input number
N2 = # ... YOUR NUMBER HERE ... # <- hidden layer number
N3 = # ... YOUR NUMBER HERE ... # <- output number

# initialize weight
W1 = np.random.random((N1, N2)) / np.sqrt(N1)
W2 = np.random.random((N2, N3)) / np.sqrt(N2)

#initialize bias
B1 = np.zeros((1, N2))
B2 = np.zeros((1, N3))
```

In the above script, you can see a new trick for the weights. Weights are randomly initialized in the $[0,1]$ range and divided by the root squared of the entry number. This trick allows to avoid saturation of the backpropagation method. This saturation is generally given by the derivation of the Sigmoid function that tends to zero for high input values. The following chart highlights this behavior.



When the result of $\sigma'(x)$ tends to zero, the backpropagation method has no effect because the correction factors Δw and Δb tend to null values.

task4. Implement the feed forward step with the mini-batch approach

task5. Implement the back propagation method using stochastic gradient descent algorithm

task6. Monitor the evolution of loss versus epochs and check your full algorithm. The loss must decrease to zero.

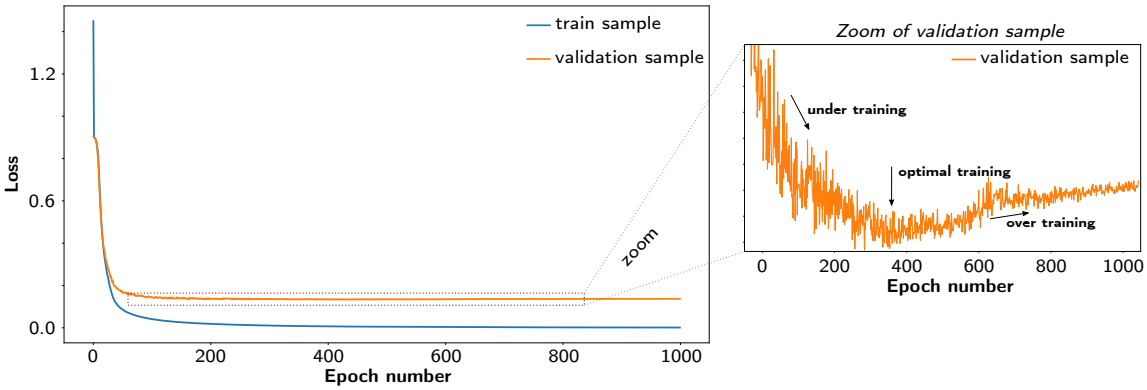
task7. Apply the trained neural network to the full data set in order to count the occurrence of good prediction. You must obtain something like 90% of correct prediction.

Here, a problem appears. The validation process involves the same data set which was used during the learning step. A better approach involves two different data set:

1. the first part of the data set is reserved to the learning process. This set is called the **training sample**.
2. The second part of the data set is reserved for evaluating the learning process. This set is called the **validation sample**.

task8. Separate your data in two different samples: the *training sample* and the *validation sample*. Monitor the losses of these two data sets.

If your code is right, you should obtain something like that.



If you take a look at the whole figure scale, the validation sample looks flat. Indeed, if you make a zoom (see the right chart), the validation sample goes down to an optimal value. This value is characterized by a minimal loss value for a given epoch number. Below this number, the neural network is *under-trained* and above this epoch number, the machine is *over-trained*. The over-trained behavior is not obvious. It should be considered as an over optimization of the weights and bias regarding the training data sample. You can observe it by zooming on the train sample: the related loss decreases continuously while the validation loss increases. Over-trained is really common in machine learning and should be proscribed. It can be detected only if you use separated data for training and validating. That's why data separation is really important.

Now, you can use this metrics to choose an optimum value of epoch, hidden layer number, mini-batch size... This step, that consist in tuning the meta parameters of a neural network is called *meta-optimization*.

> ℹ *To avoid meta-optimization overfitting, a good practice is to keep separately a third data set. This unused data sample will be devoted only for ultimate validating. It allows to prevent for too much aggressive meta-optimization.*

task9. Process to *meta-optimization* in order find an optimal set of parameters for your neural network.

You can download the correction here: `digit.py`

7 Conclusion

In this part, we have coded from scratch single layer and multi-layer neural networks. It allows us to understand the main features of ML: feed forward, back propagation, over-fitting, etc. Indeed, coding from scratch is good for understanding but not really efficient. If you need professional ML usage, my advice is to use ML modules such as *Keras* or *Scikit-learn*. With this course, that introduces the basis of ML, you are now aware about the main concepts of ML that are required to take in hand these modules. Of course, you can also continue to learn by yourself. A lot of courses and tutorial are available on the web. Thanks to this resources, you may over-fit yourself in ML!:)