Introduction to Python

Albert Ruiz Cirera

Departament de Matemàtiques

Universitat Autònoma de Barcelona

Version: March 6, 2018

Contents

Python

Python is a high-level programming language widely used in data science. Some of the properties are:

- It is open source and multi-platform.
- Emphasises the readability of the source code.
- It is also used for production of web data analytics.
- It is an interpreted language.
- It uses indentation and white spaces to delimit code blocks instead of brackets, so, spaces are important.

Among the possible installations we can either just install **Python** downloading from the project site (and install individual packages as needed) or a complete environment which comes with pre-installed libraries, like **Anaconda** (which contains **Jupyter**).

Once installed we can work either with the shell (editing and running our programs directly), with IDLE (Integrated Development and Learning Environment) or from a notebook (in a web page). In this notes we will use a `jupyter notebook` (formerly known as `iPhyton notebook`).

Currently there are two different versions of **Python** with slightly different structure in some commands: **Python 2** and **Python 3**. The syntax of the commands in this notes are prepared to be executed in **Python 3**.

1 Python 2 vs Python 3

There are currently two versions of **Python**. Somewhat confusingly, **Python 3** introduced incompatible changes to the language, so code written in **Python 2** may not work in **Python 3** and vice versa. The situation is:

- **Python 2:** There are no major releases of this version and the final one is **2.7**.
- **Python 3:** This version is under active development and the current stable version is **3.6**.

The main differences between the two versions are:

- **Python 2** (currently) has slightly better library support. This will no affect our course, as the libraries we will use (**os**, **numpy** and **pandas**) are available in both versions.
- Some commands have different syntax: in this course this affect to the `print` command. In version **2**, this is an statement, while in version **3** it is a function. Here we can see two examples.
  In **Python 2**:
  ```
  print 'Python', python_version()
  ```
  Will produce:
  ```
  Python 2.7.12
  ```
  The corresponding command in **Python 3** is:
  ```
  print('Python',python_version())
  ```

- Integer division (`/` or `//`): In **Python 2**, `/` keeps the type to `integer`, which means, for example, that `3/2` gives 1. In **Python 3**, we get `3/2`= 1.5 and integer division in **Python 3** must be done with `//`: `3//2`= 1.

- Other which do not affect to this course, related to unicode characters and the use of memory in `range` and `xrange`.

The **Python 2** package **__future__** allows to execute some commands in **Python 3** syntax (as `print`), so we can start a session or code with:

```
In [1]:  import __future__
```

2 Jupyter notebooks

We will use **Jupyter** implementation of **Python**, which enables a server that must be reached with a web browser.

Once installed we can call it with the command `jupyter notebook`, and follow the instructions to see it in a web browser.

The web browser shows a file explorer and we can navigate on the different folders and open or create files to work with.

To start with, we choose a new file to work with through the menu **New - Python3**. It opens a new tab in the browser with a Jupyter notebook. At the top of this new tab, we can edit the title, currently to **Untitled**. For further references, in this notes we call it **Session1** (this is stored in the local file **Session1.ipynb**, which can be copied/moved/sent and contains all the structure of the document).

We can use cells to execute **Python** commands or add titles, remarks, notes, ... We can decide which type of cell is with the button **Code**, where we can choose **Markdown** for the non-compiled text.

For example, we can add the following text to the first cell, which has been declared a **Markdown**:

```
# This is my first notebook
```

and run the cell with **Shift+Intro** (or the menu **Cell - Run cells**). In this command, we are using **Markdown** syntax and the most common commands are:

- **Headers:** `#` is used for main headers, `##` is used for seconday, ..., till `######` for the last one.

- **Emphasis:** italics are produced with asterisks `*` (for opening and closing): `*italics*` produce *italics*. Double asterisks `**` produce bold type fonts. In both cases, we can use underscore `_` instead of asterisk.

- **Lists:** To enumerate a list we can start the corresponding lines with a number and a point. It will not take into account the number we use in each position. For example:

```
1.  First element
3.  Second element
    Indented sentence inside the second element
```

  Will produce:

  1. First element

2. Second element

   Indented sentence inside the second element

   For unordered lists we use start the corresponding lines either with asterisks *, plus + or minus -.

- **Code:** to type code inline, we use back-ticks ` around it. If it affects a paragraph, we must type three back-tips and (optional) the language of the code:

  ```
  ```python
  s = 'Hello world'
  print(s)
  ```
  ```

- **Formulas**: we can insert formulas with dollar symbol $ and the LaTeX syntax.

Once we have some cells in a **Jupyter notebook** we can use the following short-cuts to work in it:

- With **Shift+Intro** we execute the cell.
- With **Alt+Intro** we execute the cell and open a new one after the execution.
- With **Esc** we change to command mode and we can use the arrow keys to navigate. To go inside a selected cell, press **Intro** (change to edit mode).
- In command mode, with **M** we convert the cell to **Markdown**, while with **Y** we turn it to a **Code** cell.

3 Syntax

3.1 Use of variables and functions

The assignment statement is = (the equals sign). So, we can assign a value and operate with the usual arithmetic:

```
In [1]:  r=3
In [2]:  r
In [3]:  pi=3.14159; pi
In [4]:  l=2*pi*r; l
In [5]:  A=pi*r**2; A
In [6]:  s="Hello world!"
In [7]:  s+s
In [8]:  r*s
In [9]:  pi*s
```

From these commands, we can see that we can interact with integers and floats freely[1], and that the same additive and multiplicative syntax applies for concatenating and "multiplying" strings (always by an integer).

[1] In **Python 2**, the division of integers produce an integer.

To se the value of a variable or the result of an operation, we can either ask directly as the last execution of a box, without the `;` (semicolon)[2], or use the command `print`:

```
In [10]:  print(pi);
```

In general, functions are called by the name (`print` in this case) with comma separated arguments between parenthesis (parenthesis are even needed when there are not arguments).

A short cut to multiple assignment can be done with the following syntax:

```
In [11]:  a,b,c=2.71828,"Hello",3
In [12]:  print(c,b,a);
```

To get information about a variable we can use the command `type`:

```
In [13]:  print(type(r),type(pi),type(s));
```

A variable can also contain a list. Lists are data structures which must be delimited by brackets `[]` and the comma `,` separates elements:

```
In [14]:  l1=[1,2,3,4,5];
In [15]:  print(l);
```

We can access to any position with the following syntax (take into account that the first position is indexed by 0):

```
In [16]:  print(l1[0])
In [17]:  print(l1[3])
In [18]:  print(l1[2:4])
In [19]:  print(l1[2]=99)
In [20]:  print(l1);
```

We can mix different type of elements in a list, including other lists:

```
In [21]:  l2=[l1,"Hello","world"];
In [22]:  print(l2);
```

When having a list of numbers, there are basic functions will allow us to get the sum of all the positions, which, combinated with the function `len`, we can get the mean of the values of the list:

```
In [23]:  sum(l1)
In [24]:  print('The mean is',sum(l1)/len(l1))
```

Moreover, we can add an element to the end of a list with the function `append` and remove last position with `pop`:

```
In [25]:  l1.append(15)
          print(l1)
In [26]:  l1.pop()
          print(l1)
```

Tuples are also sequences of objects in Python, like lists, but the main difference is that these objects cannot be changed. The syntax for creating tuples is comma `,` separated values inside parenthesis `()`, while reading the elements in the tuple is like it was a list.

[2]Semicolon is used to execute more than one command in the same line. In particular, ending a box with semicolon is used as an empty final command.

```
In [27]: tup1=(1,2,3,4,5)
         tup2=(100,)
         tup3=(1,2,3.14,"Hello")
         print(tup1[1])
In [28]: print(tup2[0])
In [29]: print(tup3[1:3])
```

A set is an unordered collection of comma separated distinct elements.

```
In [30]: set1={3,2,4,1,2}
In [31]: print(set1)
```

Where we can see that it does not preserve the entered order and remove repetitions.

Another type of data which will interest to us is a dictionary, which can be considered as a comma separated set of labelled values. For example:

```
In [32]: months = {'January':'Gener','February':'Febrer','March':'Març',
         'April':'Abril','May':'Maig','June':'Juny',
         'July':'Juliol','August':'Agost','September':'Setembre',
         'October':'Octubre','November':'Novembre','December':'Desembre'}
```

Access to the values uses the label. We can duplicate a dictionary, delete an entry or all the values or the dictionary as follows:

```
In [33]: print(months['April'])
In [34]: mesos=dict.copy(months) # creates a duplicate of the dictionary
In [35]: del months['May'] # remove entry with key 'May'
In [36]: mesos.clear() # remove all entries in mesos dict[3]
In [37]: del mesos # delete entire dictionary
```

Moreover, we can check the type of the variable `months` and of any of its variables:

```
In [38]: type(months)
In [39]: type(months['February'])
```

A particular case of dictionary can be used for data: each label contains a list of values, considered as a columns in a data frame. For example:

```
In [40]: data = {'name':  ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
         'age':  [42, 52, 36, 24, 73],
         'preTestScore':  [4, 24, 31, 2, 3],
         'postTestScore':  [25, 94, 57, 62, 70]}
```

3.2 Using package functions

Just the basic functions in **Python** are accessible directly when we start a session, but we can also access to a huge amount of commands making accessible particular packages.

The command `import` is used to load the packages to our session or script.

[3] In this instruction we are applying the function `clear` to the variable `mesos`.

Example 3.1

If we want to see which is our current folder or to change it, we can use `getcwd` function from the **os** package:

```
In [1]: import os
In [2]: os.getcwd()
```

All new functions loaded with this command are called as `os.name`, where `name` refers to the name of the function.

We can get information about the package just typing the imported name, and a list of the imported functions with `dir`:

```
In [3]: os
In [4]: dir(os)
```

We can change the prefix used when loading a package with the `as` option:

Example 3.2

Package **numpy** is used for scientific computing in **Python**, which is usually imported as `np`. It contains a value for π and an implementation of the $\sin$ function:

```
In [5]: import numpy as np
In [6]: print(np.pi)
In [7]: print(np.sin(np.pi))
```

We can just one (or some) functions from a package with the `from` command (in this case the function names are nor preceded by the package name):

```
In [8]: from numpy import sin,cos,pi
In [9]: print(cos(pi))
```

3.3 Conditions and loops

The possible boolean values are `True` and `False`, and we can obtain them comparing values:

Example 3.3

The following statements compare values. We can just compare values of the same type (for example, two numbers or two strings, with alphabetic order). Moreover, we can use `and` and `or` statements:

```
In [1]: 2<3
In [2]: 5<=4
In [3]: 7==7
In [4]: "Hello"=="Bye"
In [5]: "Hello"!="Bye"
In [6]: "Hello">"Bye"
In [7]: 2<3 and 5<3
```

```
In [8]:  2<3 or 5<3
In [9]:  (2<3 and 8<7) or (6<7)
```

The `if` statement allows us to execute a (block of) command(s) if some condition is fulfilled. Moreover, there is also a structure with `else` and `elif` options.

Example 3.4

Assume we want to grade with a letter (A, B, C, D or F) according to the value of score ($\geq$90, 90>score$\geq$80, 80>score$\geq$70, 70>score$\geq$60, or 60>score). In this case we can use the `if-elif` structure:

```
In [10]:  score = 68
In [11]:  if score >= 90:
              letter = 'A'
          elif score >= 80:
              letter = 'B'
          elif score >= 70:
              letter = 'C'
          elif score >= 60:
              letter = 'D'
          else:
              letter = 'F'
In [12]:  letter
```

Loops in **Python** can be done either with `for` or `while`. This will open an environment which will execute a block of lines below which have bigger indentation:

Example 3.5

We can print the first 10 Fibonacci numbers:

```
In [13]:  a0,a1=1,1
          print(a0,a1,sep="\n")
          for i in range(9):
              a0,a1=a1,a0+a1
              print(a1)
```

Exercise 3.1

Modify the previous code to obtain a list `fibo` containing the i-th Fibonacci number at position i.

When working with lists, the `for` statement has also list comprehension:

Example 3.6

We can get a list of the first 10 squares doing:

```
In [14]:  squares = [x**2 for x in range(10)]
In [15]:  squares
```

Or we can even add a condition. The next list just considers the squares of even numbers:

```
In [16]:  evensquares = [x**2 for x in range(10) if x % 2 == 0]
In [17]:  evensquares
```

The `while` statement changes a list of values by a condition, and the loop finishes when the condition is not fulfilled.

Example 3.7

The next code compute and print all the Fibonacci numbers smaller than 100:

```
In [18]:  a0,a1=1,1
          print(a0,a1,sep="\n")
          a0,a1=a1,a0+a1
          while (a1<100):
            print(a1)
            a0,a1=a1,a0+a1
```

Both `for` and `while` statements may have an `else` statement, which is executed when reaching the end of the list (in the `for` statement) or the condition do not match (in the `while` statement).

Example 3.8

The next code compute and print all the Fibonacci numbers smaller than 100:

```
In [19]:  a0,a1=1,1
          print(a0,a1,sep="\n")
          a0,a1=a1,a0+a1
          while (a1<100):
            print(a1)
            a0,a1 = a1,a0+a1
          else:  print(a1," is bigger than 100")
```

Where, in the last line, we are using that a single statement can be placed in the same line.

Also, we can finish a loop suddenly with the `break` statement.

Example 3.9

We can see in this example that the `break` finishes the loop for `x`, so it goes back to the loop for `n`:

```
In [20]:  for n in range(2, 10):
              for x in range(2, n):
                if n % x == 0:
                  print(n, 'equals', x, '*', n//x)
                  break
              else:
                print(n, 'is a prime number')
```

The `continue` statement is used to interrupt the current block in a loop and go to the next value of the index.

Example 3.10

In this example, when the number is even, print a sentence and goes to the next `num`:

```
In [21]:  for num in range(2, 10):
              if num % 2 == 0:
                print("Found an even number", num)
                continue
              print("Found a number", num)
```

3.4 Defining new functions

We can define our own functions with the command `def`. A function may contain a long program or just a simple operation. Also, arguments may have default values or even a not fixed number of them:

Example 3.11

We can define a function which return the square of a number:

```
In [1]:  def f(x):
             return(x**2);
```

We can now call the function as any other one: if we want to compute 5^2 we can write `f(5)`.

Example 3.12

We can also define functions in more than one variable, and also with default values:

```
In [2]:  def g(x=2,y=3):
             return([x,y,x+y,x*y]);
In [3]:  g();
```

```
In [4]:  g(5)
In [5]:  g(y=5)
In [6]:  g(7,8)
```

Example 3.13

We can use the option * to define functions with a variable number of arguments:

```
In [7]:  def summ(*summands):
             s=0
             for i in summands:
               s=s+i
             return(s);
In [8]:  summ(1,2,3,4)
```

The variables used in the definition of the functions (for example, s in the sum function) are local ones, and keep the values (or are undefined) outside the function. This implies that there is no conflict with the variables used in the previous pieces of code (for example i in the for statement) which keep the values globally in the notebook.

4 NumPy

4.1 Arrays and descriptive statistics with NumPy

The library **NumPy** is the fundamental package for scientific computing in Python. It provides a multidimensional array object and routines for operate on them, including mathematical, logical, manipulation, sorting, linear algebra and statistics.

We can start looking at arrays defined with this package . These arrays can be defined as nested lists:

```
In [1]:  import numpy as np
In [2]:  a = np.array([1,2,3])
         print(a.shape)
In [3]:  print(a)
In [4]:  print(a[1])
In [5]:  b = np.array([[1,2,3],[4,5,6]])
         print(b.shape)
In [6]:  print(b)
In [7]:  print(b[0,0],b[1,1])
```

Moreover, we can define directly arrays with zero coefficients, all values equals one or to a constant value:

```
In [8]:  a = np.zeros((2,3))
         print(a)
In [9]:  b = np.ones((3,2))
```

```
          print(b)
In [10]:  c = np.full((3,2),5)
          print(c)
In [11]:  print(5*b)
In [12]:  d = np.eye(3)
          print(d)
```

Assigning to another variable an array, or part of it, does not make a copy of it: both share the same memory positions, so also the values.

```
In [13]:  a = np.array([[1,2,3],[4,5,6],[7,8,9]])
          print(a)
In [14]:  b = a[:2,1:3]
          print(b)
In [15]:  b[0,0] = 15
          print(a)
```

Moreover, it is not the same considering a position as a number than as a 1×1 array. Check the difference between `c` and `d` in the following commands:

```
In [16]:  c = a[0,0]
          print(c)
In [17]:  c = 5
          print(a)
In [18]:  d = a[0:1,0:1]
          print(d)
In [19]:  d[0,0] = 5
          print(a)
```

If we want to duplicate an array we can use the `copy` function:

```
In [20]:  a = np.array([[1,2,3],[4,5,6],[7,8,9]])
          b = np.copy(a)
          c = np.copy(a[0:2,0:2])
          b[0,0] = 99
          print("a=",a)
          print("b=",b)
          print("c=",c)
```

We can use integer arrays as index to get the values of an array:

```
In [21]:  a = np.array([[1,2,3],[4,5,6],[7,8,9]])
          print(a[[0,1,0],[0,1,2]])
In [22]:  print([a[0,0],a[1,1],a[0,2]])
In [23]:  i = np.array([0,1,0])
          j = np.array([0,1,2])
          print(a[i,j])
```

We can also extract a boolean array with the indexes fulfilling some condition:

```
In [24]: print(a>4)
In [25]: print(a[a>4])
```

We can also get the minimum and maximum of each axe (empty means all the array, 0 is the columns axe, while 1 is the rows axe) with the functions `numpy.amin` and `amax`:

```
In [26]: min_of_all=np.amin(a)
         print(min_of_all)
In [27]: min_each_col=np.amin(a,0)
         print(min_each_col)
In [28]: min_each_file=np.amin(a,1)
         print(min_each_file)
In [29]: max_of_all=np.amax(a)
         print(max_of_all)
In [30]: max_each_col=np.amax(a,0)
         print(max_each_col)
In [31]: max_each_file=np.amax(a,1)
         print(max_each_file)
```

Exercise 4.1

The `ptp` function has the same syntax as `amin` and returns the range (maximum-minimum) of the values.

(a) Compute the range of the array `a` and the range of each column.

(b) Define a new function (using `amax` and `amin`) called `the_range` with the same syntax and result as `ptp`.

NumPy has a `percentile` function (that is, give the value below which a given percentage of the observations fall). It is applied also to arrays and needs two mandatory arguments (the array and the percentile, scaled 0-100) and an optional one (the axis). A particular case is the `median`, which provides percentile 50.

```
In [32]: print(np.percentile(a,50))
In [33]: print(np.median(a))
In [34]: print(np.percentyle(a,50, axis = 1))
```

We can also, with the same syntax, compute the mean using the `mean` in the **NumPy** package:

```
In [35]: print(np.mean(a))
In [36]: print(np.mean(a, axis = 1))
```

A generalization of the mean is the average: computing the mean of an array where each position may have different weight. For example, if we have the list of scores $[7, 2, 8, 9]$ and we want to get the average corresponding to the weights $[3, 2, 4, 3]$, we must compute:

$$\frac{3*7+2*2+4*8+3*9}{3+2+4+3} = 7\,.$$

If we want to do this using **NumPy** we can use the average function:

```
In [37]: scores=np.array([7,2,8,9])
         w=np.array([3,2,4,3])
         print(np.average(scores,weights=w))
```

Remark that in multiple dimensional arrays we can also specify the axis.

std function in **NumPy** returns the standard deviation of the corresponding array (which can also be done by axes), while the var provides the variance:

```
In [38]: scores=[7,2,8,9]
         print('Standard deviation',np.std(scores))
         print('Variance',np.var(scores))
         print('Square of std',np.std(scores)**2)
```

4.2 Loading and manipulating data with NumPy

In this section we will have to access to local files which contain datasets. One of the things that we must have under control is the folder where the data is, and the folder where we are working. To get or change this information we can use the **os** package with the commands getcwd or chdir.

Example 4.1

Assume that we have the home folder **/home/user** with a folder called **Python** inside it, and we want to work in this folder.

```
In [1]: import os
In [2]: os.getcwd() # assume we get /home/user
In [3]: os.chdir('Python')
In [4]: os.getcwd()
```

This sections uses files in `http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic)` that we have downloaded locally, and which contains a file called **wdbc.data** with comma separated values. The first column corresponds to the Id of the patient, the second column to the diagnosis (B=benign, M=malign) and the 30 other columns corresponds to 10 different parameters (radius, texture, perimeter, area, smoothness, compactness, concavity, concave points, symmetry and fractal dimension) and for each one we have the mean (10 first columns), standard deviation (next 10) and maximum (last 10).

Numpy arrays must have all the values of the same type (either integer, float, string, ...). As this file contains a column corresponding to a text (B for benign and M for malign) we read it firs as a text (dtype='U' in the next command), then recode the second column to a number (0 for benign and 1 for malign) and the we convert the array to float:

```
In [5]: import numpy as np
        wdbctext=np.genfromtxt('wdbc.data',delimiter=',',dtype='U')[4]
```

[4]We can also download it directly from a url by doing (in a single line)

```
          print(wdbctext)
In [6]:   diagnosis = wdbctext[:,1]
          diagnosis[diagnosis=='B']='0'
          diagnosis[diagnosis=='M']='1'
          print(wdbctext)
In [7]:   wdbc = wdbctext.astype(float)
          print(wdbc)
```

To record all this modifications, we can save a copy of this array to a file with the command `savetxt`:

```
In [8]:   np.savetxt('wdbc_modified.txt',wdbc,delimiter=',')
```

We can delete a part of the array with the `delete` function. In this case, we work on the text array as example to avoid confusion: assume we want to delete the first column. In the next execution, the `0` corresponds to the numbering of rows or columns we want to delete (starts at 0), and parameter `1` corresponds to column (we should use `0` for rows)

```
In [9]:   wdbctext=np.delete(wdbctext,0,1)
          print(wdbctext)
```

As now me may consider that the last 30 columns of the array are divided in 3 groups of 10 columns, we take the diagnosis and first group (corresponding to the mean of the 10 variables) with a new array's name `wdbc_m`:

```
In [10]:  wdbc_m=wdbc[:,1:11]; print(wdbc_m)
```

Now we can obtain the basic statistics of each column:

```
In [11]:  print(np.mean(wdbc_m,0))
In [12]:  print(np.median(wdbc_m,0))
In [13]:  print(np.amin(wdbc_m,0))
In [14]:  print(np.amax(wdbc_m,0))
```

We can also approximate a points cloud by a polynomial. So, a particular case, is approximating by a degree one polynomial (straight line). The function `polyfit` gives as an array with the coefficients:

```
In [15]:  fit = np.polyfit(wdbc[:,2],wdbc[:,2],1)
          print(fit)
```

Getting the array `[0.39516975, 13.70697016]`. Here, the i-th position right to left (position $-i$) corresponds to the coefficient of x^i in the polynomial in x, starting with $i = 0$. So, this corresponds to the equation:

$$\text{texture} = 0.39516975 * \text{radius} + 13.70697016\,.$$

The function `poly1d` can convert a list of coefficients to a polynomial function:

```
In [16]:  regr = np.poly1d(fit)
          print(regr(0))
In [17]:  print(regr(1)-regr(0))
```

We can also do a direct computation:

```
In [18]:  X=np.ones((wdbc.shape[0],2))
```

`wdbctext=np.genfromtxt('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data',delimiter=',',dtype='U')`

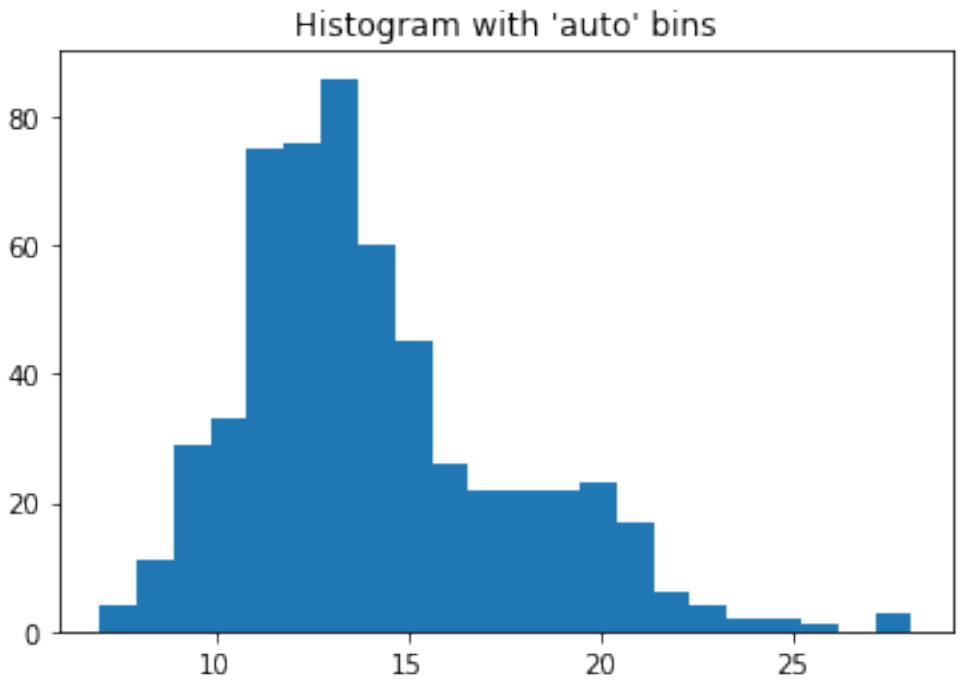
```
X[:,1]=wdbc[:,2]
Y=wdbc[:,3]
XtX=np.transpose(X).dot(X)
XtY=X.T.dot(Y) # function .T is also the transpose
print(np.linalg.inv(XtX).dot(XtY))
```

5 Graphics with matplotlib.pyplot

If we want to get graphical output from our data stored as an array, one option is to use the package **matplotlib.pyplot**. As we are working with **wdbc** data, we recall the commands for the following example:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        wdbc=np.genfromtxt("wdbc_modified.data", delimiter=",")
        plt.hist(wdbc[:,2], bins='auto')
        plt.title("Histogram with 'auto' bins")
        plt.show()
```

Obtaining:



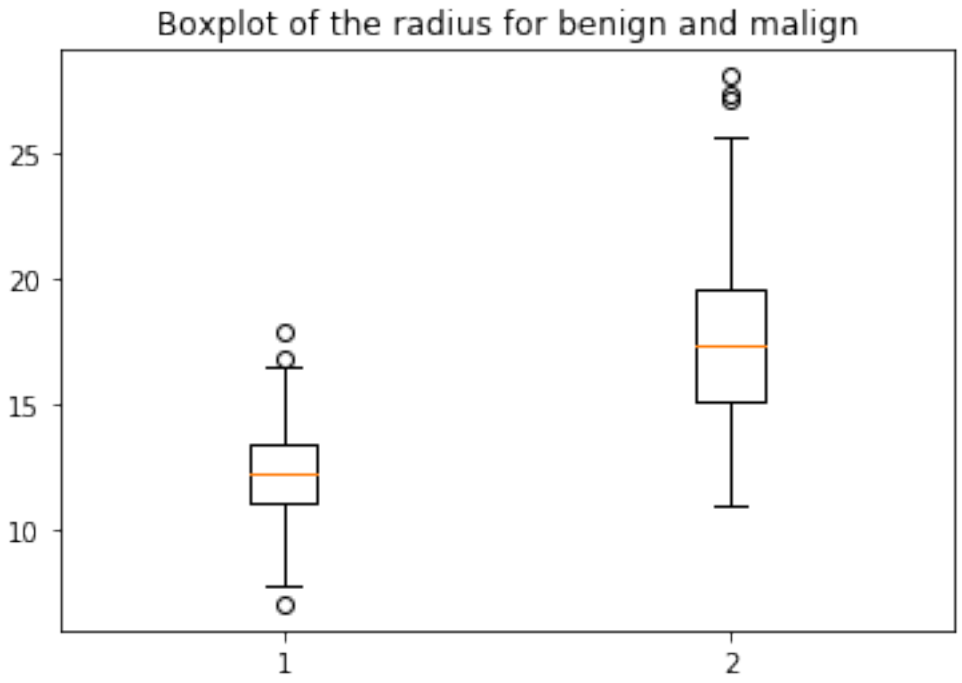
If we want to save it instead of showing the result, we can use the command `savefig`:

```
In [2]: plt.hist(wdbc[:,2], bins='auto')
        plt.title("Histogram with 'auto' bins")
        plt.savefig('histogram.png')
```

We can see in the previous list of commands that we construct the plot step by step: `plt.hist` constructs the histogram, `plt.title` adds a title and finally `plt.show` shows the result. This is the way we will work here, and, when necessary, adding more parts to he final plot.

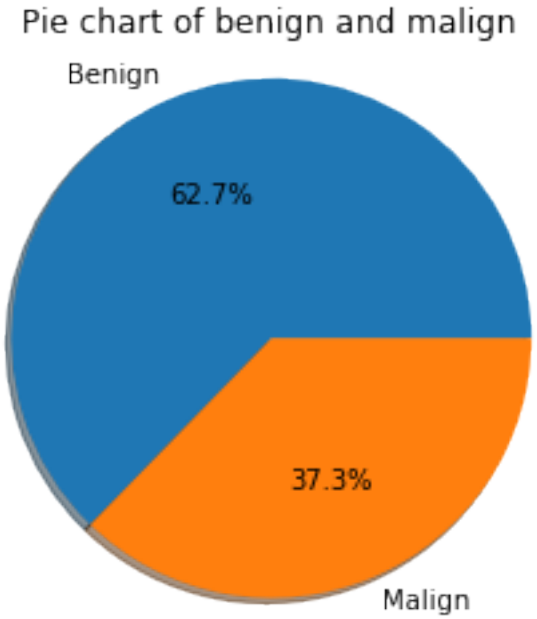
We can also use the same package to obtain a boxplot. In this case, we want a box for each case in the diagnosis (column 0):

```
In [3]: benign,malign=wdbc[wdbc[:,1]==0],wdbc[wdbc[:,1]==1]
        plt.boxplot([benign[:,2],malign[:,2]])
        plt.title("Boxplot of the radius for benign and malign")
        plt.show()
```



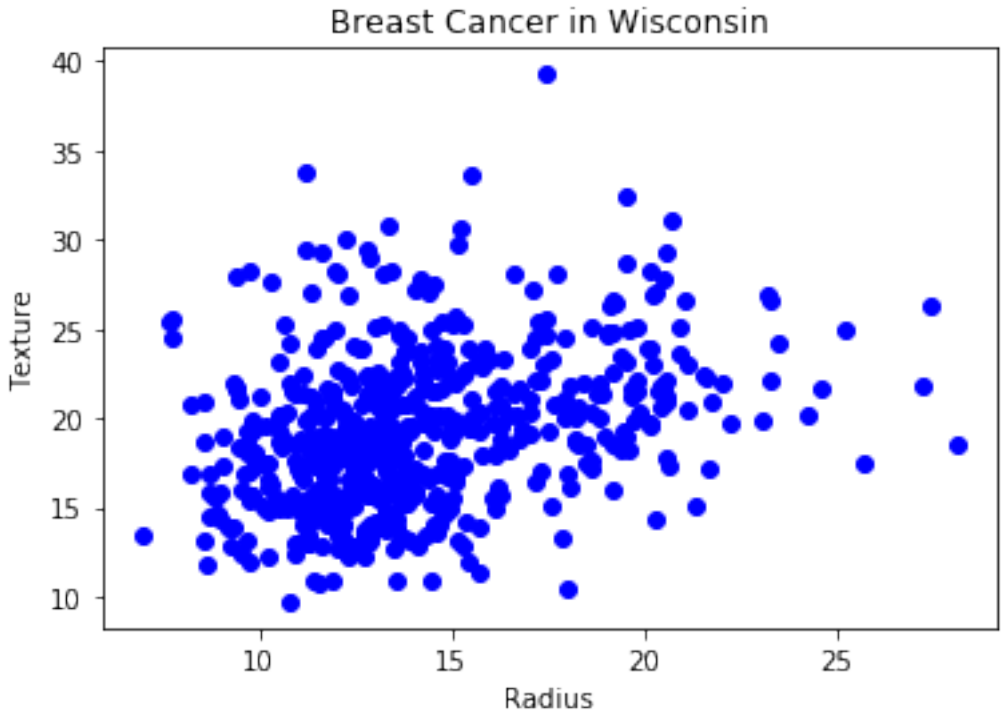
To finish with these univariate examples, we can plot a pie chart of the variable diagnosis (column 0):

```
In [4]: values,freq=np.unique(wdbc[:,1],return_counts=True)
        plt.pie(freq,labels=['Benign','Malign'],autopct='%1.1f%%', shadow=True)
        plt.title("Pie chart of benign and malign")
        plt.axis('equal')
        plt.show()
```



We can also do an scatter plot of two numerical variables. In this example, we choose the radius and the texture in the **wdbc** data:

```
In [5]: plt.plot(wdbc[:,2],wdbc[:,3],"ob")
        plt.title('Breast Cancer in Wisconsin')
        plt.xlabel('Radius')
        plt.ylabel('Texture')
        plt.show()
```



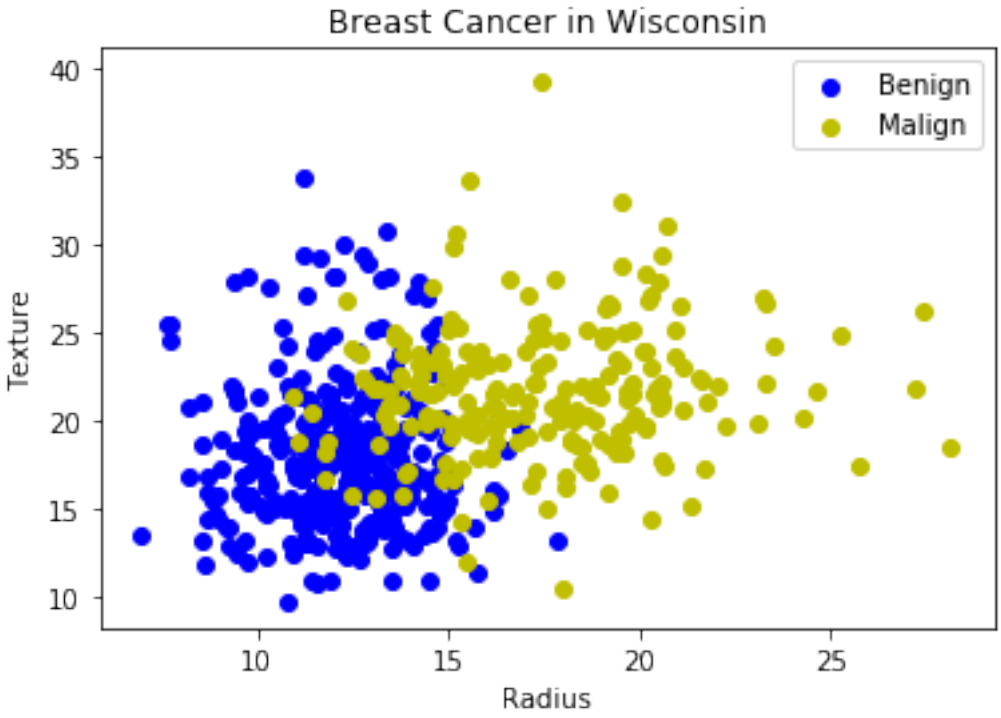
In this case, option `ob` stands for the symbol circle (`o`) in blue (`b`) as a marker. In Table 1, there is a list of options for markers and colours.

Marker		Color	
`.`	Point	`b`	Blue
`,`	Pixel	`b`	Blue
`o`	Circle	`g`	Green
`v`	Triangle down	`r`	Red
`^`	Triangle up	`c`	Cyan
`<`	Triangle left	`m`	Magenta
`<`	Triangle right	`y`	Yellow
`8`	Octagon	`k`	Black
`s`	Square	`w`	White
`p`	Pentagon	`(a,b,c)`	RGB color with $0 \leq$a,b,c≤ 1
`P`	Plus (filled)	`'a'`	$0 \leq$a≤ 1 for grey level
`*`	Star		
`h`	Hexagon 1		
`H`	Hexagon 2		
`+`	Plus		
`x`	x		
`X`	x (filled)		
`D`	Diamond		
`d`	Thin diamond		

Table 1: Examples of markers and colors

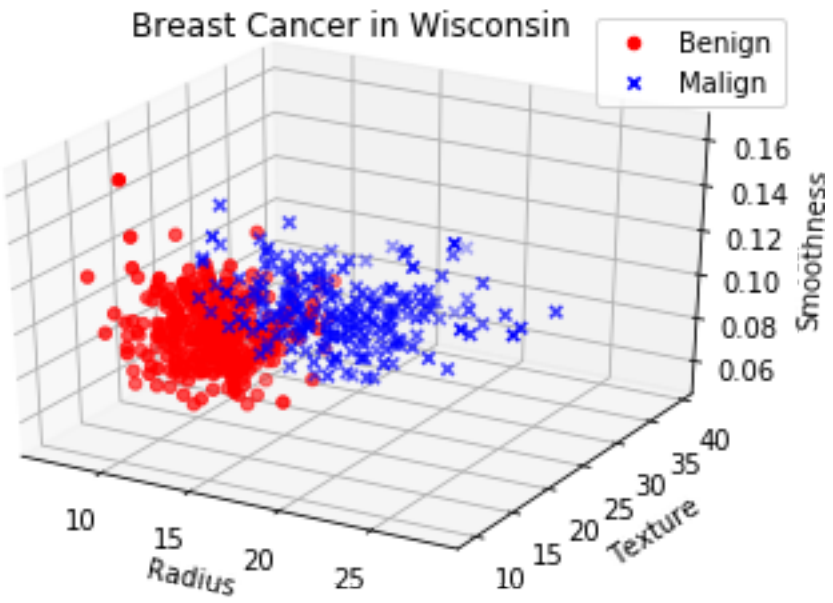
We can compose a plot with different plots adding them before executing `show` command. In the next example, we also use function `scatter` in **matplotlib.pyplot** package:

```
In [6]: benign,malign=wdbc[wdbc[:,1]==0],wdbc[wdbc[:,1]==1]
        benignplot=plt.scatter(benign[:,2],benign[:,3],color='b')
        malignplot=plt.scatter(malign[:,2],malign[:,3],color='y')
        leg = plt.legend([benignplot,malignplot], ['Benign','Malign'])
        plt.title('Breast Cancer in Wisconsin')
        plt.xlabel('Radius')
        plt.ylabel('Texture')
        plt.show()
```



We can even do 3d scatter plots:

```
In [7]:  from mpl_toolkits.mplot3d import Axes3D
In [8]:  fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
         benignplot=ax.scatter(benign[:,2],benign[:,3],benign[:,6],c='r', marker='o')
         malignplot=ax.scatter(malign[:,2],malign[:,3],malign[:,6],c='b', marker='x')
         leg = plt.legend([benignplot,malignplot], ['Benign','Malign'])
         plt.title('Breast Cancer in Wisconsin')
         ax.set_xlabel('Radius')
         ax.set_ylabel('Texture')
         ax.set_zlabel('Smoothness')
         plt.show()
```



6 pandas

pandas (**pan**el **da**ta) is a Python library for data analysis. It offers a number of data exploration, cleaning and transformation operations that are critical in working with data in Python.

pandas build upon **numpy** and **scipy**, providing easy-to-use data structures and data manipulation functions with integrated indexing.

The main data structures pandas provides are `Series` and `DataFrames`. After a brief introduction to these two data structures and data ingestion, the key features of pandas shown in this section are:

- Generating descriptive statistics on data.
- Data cleaning using built in **pandas** functions.
- Frequent data operations for subsetting, filtering, insertion, deletion and aggregation of data.
- Merging multiple datasets using dataframes.

6.1 Basics on pandas

One of the objects for dealing with data are `Series` , which are one dimensional labelled arrays which can hold any data type (integers, strings, floating point numbers, Python objects, ...). The axis labels are known as the `index`.

Example 6.1

Here we create a `serie` manually:

```
In [1]:  import pandas as pd
```

```
In [2]: ser=pd.Series([10,'foo',30,'bar',50],index=['tom','bob','nancy','dan','eric'])
        print(ser)
```

We can ask information about this object:

```
In [3]: print(type(ser))   In [4]: print(ser.index)
In [5]: print(type(ser.index))
In [6]: print(ser.shape)
In [7]: print(ser.shape[0])
```

And also extract or copy to another variable some data or part of a serie:

```
In [8]: print(ser['bob']) # The result is a string
In [9]: sernb=ser[['nancy','bob']] # The result is a Serie
In [10]: sernb=ser.loc[['nancy','bob']]
         print(sernb)
In [11]: sernb['bob']=60
         print(ser)
In [12]: print(ser[0])
In [13]: print(ser(:2))
In [14]: print(ser.iloc(:2))
```

The main object for dealing with data are `DataFrames`: a `DataFrame` is a two dimensional labelled array. We can also think as a collection of `Series` organized as columns.

`DataFrames` can be created from dictionaries of `series` or `lists`:

Example 6.2

Here we define a `DataFrame` as a dictionary with two `Series`:

```
In [15]: d ={'one':pd.Series([100.,200.,300.],index=['apple','ball','clock']),
         'two':pd.Series([111.,222.,333.],index=['apple','ball','ring'])}
In [16]: df=pd.DataFrame(d)
In [17]: print(df)
```

Other options that fit when definig a `DataFrame` from a dictionary is to select some of the cases or columns (here we just see the result, without assigning to any variable):

```
In [18]: pd.DataFrame(d, index=['dancy', 'ball', 'apple'], columns=['two','five'])
```

Now we can see which is the type of this object (`DataFrame`) and of each column (`Serie`):

```
In [19]: print(type(df))
In [20]: print(df['one'])
In [21]: print(type(df['one']))
```

Also information about its structure:

```
In [22]: print(df.shape)
In [23]: print(df.index)
In [24]: print(df.columns)
```

Example 6.3

This example differs from the first one in the structure we fill the data. We just define the data from two dictionaries and see the result of each command:

```
In [25]: data = ['alex': 1, 'joe': 2, 'ema': 5, 'dora': 10, 'alice': 20]
In [26]: pd.DataFrame(data)
In [27]: pd.DataFrame(data, index=['orange', 'red'])
In [28]: pd.DataFrame(data, columns=['joe', 'dora','alice'])
```

We finish this section with some basic operation on the `DataFrame df` defined previously:

```
In [29]: df['three']=df['one']*df['two']
         print(df)
In [30]: df['flag']=df['one']>250
         print(df)
In [31]: three = df.pop('three')
         print(three)
In [32]: print(df)
In [33]: del df['two']
In [34]: df.insert(1,'oneagain',df['one])
In [35]: print(df)
```

6.2 Loading and manipulating data with pandas

This section will also load local files, so we have to control the folder where we are working. We recommend to see Example 4.1 for more details.

The library **pandas** allows us to load tables as data-frames. A data-frame is the equivalent of a spreadsheet table, and contains named columns which may be of different data type. In the next example we will load the file **diabetes.csv**, which contains 9 variables separated by comma and the not available values are codified by a point:

```
In [1]: import pandas as pd
In [2]: data = pd.read_csv('diabetes.csv', sep=',')[5]
In [3]: data
```

So we see the value of `data`.

If we want to get more information, we can access to the details with the instructions `shape` and `columns`:

```
In [4]: data.shape
In [5]: data.columns
In [6]: data.index
In [7]: data.head()
```

[5] If you prefer not to download it and deal with the folder where it is, you can use a hidden copy from my webpage: `data = pd.read_csv('http://mat.uab.cat/ albert/tmp/diabetes.csv', sep=',')`

With the command `iloc` we can access to a subset of the data frame:

```
In [8]:  subdata1 = data.iloc[[0,10,40,500]] # the result is a data frame
In [9]:  subdata2 = data.iloc[1] # the result is a serie
```

Once we know the names of the columns, we can access by the name, use them to filter the results (get a subset) or to get any statistics:

```
In [10]:  data['Age']
In [11]:  data[data['Age']<30]
In [12]:  data['Age'].mean()
In [13]:  data['Age'].min()
In [14]:  data['Age'].max()
In [15]:  data['Age'].std()
In [16]:  data['Age'].mode()
In [17]:  data['Age'].describe()
```

We can also ask information for all the data frame:

```
In [18]:  data.describe()
In [19]:  data.corr()
```

Filter variables al constructed assigning the result of a condition:

```
In [20]:  filter = data['Age]<30
In [21]:  filter.any()
In [22]:  filter.all()
```

We can look for null values in the data frame:

```
In [23]:  data.isnull().any()
In [24]:  data.loc[100,'Age'] = None
In [25]:  data.isnull().any()
```

And drop the corresponding rows:

```
In [26]:  cleandata = data.dropna()
In [27]:  cleandata.isnull().any()
```

We can obtain statistics of the data frame divided in different groups:

```
In [28]:  data.groupby('Pregnacies').count()
In [29]:  data.groupby([data['Pregnacies'],data[Outcome]]).mean()
```

Finally, we can also convert intervals in discrete values:

```
In [30]:  data['Age_cat']=pd.cut(data['Age'],bins=[0,25,60,200],
          labels=['Young','Adult','Old'])
          data.groupby('Age_cat').count()['Age']
```

Index