This document is a brief introduction to Python, the programming language we will use in the lab sessions, and to Matplotlib and Numpy, two modules for plotting and scientific computing in Python, respectively.

Note: this document corresponds to the introduction to python given at the Lisbon Machine Learning School in July 2013 (`http://lxmls.it.pt/`).

0.1 Python

0.1.1 Python Basics

Running Python code

We will start by creating and running a dummy program in Python which simply prints the "Hello World!" message to the standard output (this is usually the first program you code when learning a new programming language).

There are two main ways in which you can run code in Python:

From a file – Create a file named `yourfile.py` and write your program in it, using your favorite text editor:

```
print 'Hello World!'
```

After saving and closing the file, you can run your code by calling:

```
python yourfile.py
```

in the command line. This will run the program and display the message "Hello World!". After, the control will return to the command line.

In the interactive command line – Start the interactive command line in Python using the command `python`. After this, you can run Python code by simply writing it and pressing enter. In our lab sessions, we will use Python in interactive mode several times. The standard Python interface is not very friendly, though. IPython, which stands for *interactive Python*, is an improved Python shell. It saves your command history between sessions, has basic auto-complete, and has internal support for interacting with graphs through matplotlib. IPython is also designed to facilitate running parallel code on clusters of machines, but we will not make use of that functionality.

To run IPython, simply type `ipython` on your command line[1]. For interactive numeric use, the `--pylab` flag imports numpy and matplotlib (the two libraries we will extensively use in the lab sessions) for you and sets up interactive graphs:

```
IPython --pylab
```

You can then run Python commands in the IPython command line

```
 In[]: print "Hello, World!"
Out[]: Hello, World!
```

but you can also run Python code written into a file.

```
 In[]: run ./yourfile.py
Out[]: Hello, World!
```

Keep in mind that you can easily switch between these two modes. You can quickly test commands in the command line directly and e.g. inspect variables. Larger sections of code can be stored and run from files.

[1] Note that in some systems, e.g. Linux, you may need to run the command lower-cased.

Help and Documentation

There are several ways to get help on IPython:

- Adding a question mark to the end of a function or variable and pressing Enter brings up associated documentation. Unfortunately, not all packages are well documented. Numpy and matplotlib are pleasant exceptions;
- `help('print')` gets the online documentation for the `print` keyword;
- `help()`, enters the help system.
- When at the help system, type `q` to exit.

For more information on IPython (Pérez and Granger, 2007), check the website: `http://ipython.scipy.org/moin/`

Exiting

Exit IPython by typing `exit()` or `quit()` (or typing CTRL-D).

0.1.2 Python by Example

Data Structures

In Python, you can create lists of items with the following syntax:

```
countries = ['Portugal','Spain','United Kingdom']
```

A string should be surrounded with apostrophes (’). You can access a list with the following:

- `len(L)`, which returns the number of items in L;
- `L[i]`, which returns the item at index i (the first item has index 0);
- `L[i:j]`, which returns a new list, containing all the items between indexes i and $j-1$, inclusive.

Exercise 0.1 *Use L[i:j] to return the countries in the Iberian Peninsula.*

Loops and Indentation

A loop allows a section of code to be repeated a certain number of times, until a stop condition is reached. For instance, when the list you are iterating has reached its end or when a variable has reached a certain value (in this case, you should not forget to update the value of that variable inside the code of the loop). In Python you have `while` and `for` loop statements. The following two example programs output exactly the same using both statements: the even numbers from 2 to 8.

```
i = 2
while i < 10:
  print i
  i += 2
```

```
for i in range(2,10,2):
    print i
```

You can copy and run this from the IPython command line. Alternatively you can write this into your `yourfile.py` file an run it as well. Notice something? It is possible that the code did not act as expected or maybe an error message popped up. This brings us to an important aspect of Python: **indentation**. Indentation is the number of blank spaces at the leftmost of each command. This is how Python differentiates between blocks of commands inside and outside a statement, e.g. `while`, `for` or other. All commands within a statement have the same number of blank spaces at their leftmost. For instance, consider the following code:

```
a=1
while a <= 3:
    print a
    a += 1
```

and its output:

```
1
2
3
```

Exercise 0.2 *Can you then predict the output of the following code?:*

```
a=1
while a <= 3:
    print a
a += 1
```

Bear in mind that indentation is often the main source of errors when starting to work with Python. Try to get used to it as quickly as possible. It is also recommendable that you use a text editor that can display all characters e.g. blank space, tabs, since these characters can be visually similar but are considered different by Python. One of the most common mistakes by newcomers to Python is to have their files indented with spaces on some lines and with tabs on other lines. Visually it might appear that all lines have proper indentation, but you will get an `IndentationError` message if you try it.

Control Flow

The `if` statement allows to control the flow of your program. The next program outputs a greeting that depends on the time of the day.

```
hour = 16
if hour < 12:
    print 'Good morning!'
elif hour >= 12 and hour < 20:
    print 'Good afternoon!'
else:
    print 'Good evening!'
```

Functions

A function is a block of code that can be reused to perform a similar action. The following is a function in Python.

```
def greet(hour):
    if hour < 12:
        print 'Good morning!'
    elif hour >= 12 and hour < 20:
        print 'Good afternoon!'
    else:
        print 'Good evening!'
```

You can write this command into IPython interactive command line directly or write them into a file and run the file in IPython. Once you do this the function will be available for you to use. Call the function `greet` with different hours of the day (for example, type `greet(16)`) and see that the program will greet you accordingly.

Exercise 0.3 *Note that the previous code allows the hour to be less than 0 or more than 24. Change the code in order to indicate that the hour given as input is invalid. Your output should be something like:*

```
greet(50)
Invalid hour: it should be between 0 and 24.
greet(-5)
Invalid hour: it should be between 0 and 24.
```

Profiling

If you are interested in checking the performance of your program, you can use the command `%prun` in IPython (this is an IPython-only feature). For example:

```
def myfunction(x):
    ...

%prun myfunction(22)
```

The output of the `%prun` command will show the following information for each function that was called during the execution of your code:

- `ncalls`: The number of times this function was called. If this function was used recursively, the output will be two numbers; the first one counts the total function calls with recursions included, the second one excludes recursive calls.
- `tottime`: Total time spent in this function, excluding the time spent in other functions called from within this function.
- `percall`: Same as `tottime`, but divided by the number of calls.
- `cumtime`: Same as `tottime`, but including the time spent in other functions called from within this function.
- `percall`: Same as `cumtime`, but divided by the number of calls.
- `filename:lineno(function)`: Tells you where this function was defined.

Debugging in Python

During the lab sessions we will use the previously described IPython iterative command line which allows you to execute a script, command by command. This should limit the need for debugging tools. However, there will be situations in which we will use and extend modules that involve more elaborated code and statements, like classes and nested functions. Although desirable, it should not be necessary for you to fully understand the whole code to carry out the exercises. It will suffice to understand the algorithm as explained in the theoretical part of the class and the local context of the part of the code where we will be working.

The simplest way to do this is to run the code and stop the execution at a given point (called break-point) to get a quick glimpse of the variable structures and to inspect the execution flow of your program. For that, you can use the ipdb module.

In the following example, we use this module to inspect the `greet` function:

```
def greet(hour):
    if hour < 12:
        print 'Good morning!'
    elif hour >= 12 and hour < 20:
        print 'Good afternoon!'
    else:
        import ipdb;ipdb.set_trace()
        print 'Good evening!'
```

Load the new definition of the function into IPython by writting this code in a file and running it. Now, if you try `greet(50)` the code execution should stop at the place where you located the break-point (that is, in the `print 'Good evening!'` statement). You can now run new commands or inspect variables. For this purpose there are a number of commands you can use. The complete list can be found at `http://docs.python.org/library/pdb.html`, but we provide here a short table with the most useful:

(h)elp	Starts the help menu
(p)rint	Prints a variable
(p)retty(p)rint	Prints a variable, with line break (useful for lists)
(n)ext line	Jumps to next line
(s)tep	Jumps inside of the function we stopped at
c(ont(inue))	Continues execution until finding breakpoint or finishing
(r)eturn	Continues execution until current function returns
b(reak) n	Sets a breakpoint in in line n
l(ist) [n], [m]	Prints 11 lines around current line. Optionally starting in line n or between lines n, m
w(here)	Shows which function called the function we are in, and upwards (stack[2])
u(p)	Goes one level up the stack (frame of the function that called the function we are on)
d(down)	Goes one level down the stack
blank	Repeat the last command
expression	Executes the python expression as if it was in current frame

Table 1: Basic pdb/ipdb commands, parentheses indicates abbreviation

So getting back to our example, we can type n(ext) once to execute the line we stopped at

```
ipdb> n
> ./lxmls-toolkit/yourfile.py(8)greet()
      7                 import ipdb;ipdb.set_trace()
----> 8                 print 'Good evening!'
```

Now we can inspect the variable `hour` using the p(retty)p(rint) option

```
ipdb> pp hour
50
```

From here we could keep advancing with the n(ext) option or set a b(reak) point and type c(ontinue) to jump to a new position. We could also execute any python expression which is valid in the current frame (the function we stopped at). This is particularly useful to find out why code crashes, as we can try different alternatives without the need to restart the code again.

Exercise 0.4 *Use the debugger to debug the `buggy.py` script which attempts to repeatedly perform the following computation:*

1. *Start* $x_0 = 0$
2. *Iterate*
   (a) $x'_{t+1} = x_t + r$, *where* r *is a random variable.*
   (b) *if* $x'_{t+1} >= 1.$, *then stop.*
   (c) *if* $x'_{t+1} <= 0.$, *then* $x_{t+1} = 0$
   (d) *else* $x_{t+1} = x'_{t+1}$.
3. *Return the number of iterations.*

This is attempting to predict the number of times that a "drunk walker" (i.e., an agent who takes random steps to the left or to the right) takes to go from 0 to 1.

Having repeated this computation a number of times, the program prints the average. Unfortunately, the program has a few bugs, which you need to fix.

[2]Note that since we are inside the IPython command line, the IPython functions will also appear at the top.

0.1.3 Exceptions

Occasionaly, a syntactically correct code statement may produce an error when an attempt is made to execute it. These kind of errors are called *exceptions* in Python. For example, try executing the following:

```
10/0
```

A *ZeroDivisionError* exception was raised, and no output was returned. Exceptions can also be forced to occur by the programmer, with customized error messages (for a complete list of built-in exceptions, see `http://docs.python.org/2/library/exceptions.html`).

```
raise ValueError("Invalid input value.")
```

Exercise 0.5 *Rewrite the code in Exercise 0.3 in order to raise a ValueError exception when the hour is less than 0 or more than 24.*

Handling of exceptions is made with the *try* statement:

```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "Oops! That was no valid number. Try again..."
```

It works by first executing the *try* clause. If no exception occurs, the *except* clause is skipped; if an exception does occur, and if its type matches the the exception named in the *except* keyword, the except clause is executed; otherwise, the exception is raised and execution is oborted (if it is not caught by outer *try* statements).

Extending basic Functionalities with Modules

In Python you can load new functionalities into the language by using the `import`, `from` and `as` keywords. For example we can load the numpy module as

```
import numpy as np
```

then we can run the following on the IPython command line

```
np.var?
np.random.normal?
```

The import will make the numpy tools available through the alias `np`. This shorter alias prevents the code from getting too long if we load lots of modules. The first command will display the help for the method `numpy.var` using the previously commented symbol `?`. Note that in order to display the help you need the full name of the function including the module name or alias. Modules have also submodules that can be accessed the same way, as shown in the second example.

0.1.4 Matplotlib – Plotting in Python

Matplotlib[3] is a plotting library for Python. It supports 2D and 3D plots of various forms. It can show them interactively or save them to a file (several output formats are supported).

[3] `http://matplotlib.org/`

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

X = np.linspace(-4, 4, 1000)

plt.plot(X, X**2*np.cos(X**2))
plt.savefig("simple.pdf")
```

Exercise 0.6 *Try running the following on IPython, which will introduce you to some of the basic numeric and plotting operations.*

```
# This will import the numpy library
# and give it the np abbreviation
import numpy as np

# This will import the plotting library
import matplotlib.pyplot as plt

# Linspace will return 1000 points,
# evenly spaced between -4 and +4
X = np.linspace(-4, 4, 1000)

# Y[i] = X[i]**2
Y = X**2

# Plot using a red line ('r')
plt.plot(X, Y, 'r')

# arange returns integers ranging from -4 to +4
# (the upper argument is excluded!)
Ints = np.arange(-4,5)

# We plot these on top of the previous plot
# using blue circles (o means a little circle)
plt.plot(Ints, Ints**2, 'bo')

# You may notice that the plot is tight around the line
# Set the display limits to see better
plt.xlim(-4.5,4.5)
plt.ylim(-1,17)
plt.show()
```

0.1.5 Numpy – Scientific Computing with Python

Numpy[4] is a library for scientific computing with Python.

Multidimensional Arrays

The main object of numpy is the multidimensional array. A multidimensional array is a table with all elements of the same type and can have several dimensions. Numpy provides various functions to access and manipulate multidimensional arrays. In one dimensional arrays, you can index, slice, and iterate as you can with lists. In a two dimensional array M, you can use perform these operations along several dimensions.

- M[i,j], to access the item in the i^{th} row and j^{th} column;
- M[i:j,:], to get the all the rows between the i^{th} and $j-1^{th}$;
- M[:,i], to get the i^{th} column of M.

[4] http://www.numpy.org/

Again, as it happened with the lists, the first item of every column and every row has index 0.

```
import numpy as np
A = np.array([
    [1,2,3],
    [2,3,4],
    [4,5,6]])

A[0,:] # This is [1,2,3]
A[0] # This is [1,2,3] as well

A[:,0] # this is [1,2,4]

A[1:,0] # This is [ [2], [4] ]. Why?
        # Because it is the same as A[1:n,0] where n is the size of the array.
```

Mathematical Operations

There are many helpful functions in numpy. For basic mathematical operations, we have `np.log`, `np.exp`, `np.cos`,... with the expected meaning. These operate both on single arguments and on arrays (where they will behave element wise).

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

X = np.linspace(0, 4 * np.pi, 1000)
C = np.cos(X)
S = np.sin(X)

plt.plot(X, C)
plt.plot(X, S)
```

Other functions take a whole array and compute a single value from it. For example, `np.sum`, `np.mean`,... These are available as both free functions and as methods on arrays.

```
import numpy as np

A = np.arange(100)

# These two lines do exactly the same thing
print np.mean(A)
print A.mean()

C = np.cos(A)
print C.ptp()
```

Exercise 0.7 *Run the above example and lookup the `ptp` function/method (use the `?` functionality in IPython).*

Exercise 0.8 *Consider the following approximation to compute an integral*

$$\int_0^1 f(x)dx \approx \sum_{i=0}^{999} \frac{f(i/1000)}{1000}.$$

Use numpy to implement this for $f(x) = x^2$. You should not need to use any loops. Note that integer division in Python 2.x returns the floor division (use floats – e.g. $5.0/2.0$ – to obtain rationals). The exact value is $1/3$. How close is the approximation?

0.2 Essential Linear Algebra with Python

Linear Algebra provides a compact way of representing and operating on sets of linear equations.

$$\begin{aligned} 4x_1 \quad -5x_2 \quad &= -13 \\ -2x_1 \quad +3x_2 \quad &= 9 \end{aligned}$$

This is a system of linear equations in 2 variables. In matrix notation we can write the system more compactly as

$$Ax = b$$

with

$$A = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, \; b = \begin{bmatrix} -13 \\ 9 \end{bmatrix}$$

0.2.1 Notation

We use the following notation:

- By $A \in \mathbb{R}^{m \times n}$, we denote a **matrix** with m rows and n columns, where the entries of A are real numbers.
- By $x \in \mathbb{R}^n$, we denote a **vector** with n entries. A vector can also be thought of as a matrix with n rows and 1 column, known as a **column vector**. A **row vector** — a matrix with 1 row and n columns is denoted as x^T, the transpose of x.
- The ith element of a vector x is denoted x_i

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Exercise 0.9 *In the rest of the school we will represent both matrices and vectors as numpy arrays. You can create arrays in different ways, one possible way is to create an array of zeros.*

```
import numpy as np
m = 3
n = 2
a = np.zeros([m,n])
print a
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

You can check the shape and the data type of your array using the following commands:

```
print a.shape
(3, 2)
print a.dtype.name
float64
```

*This shows you that "a" is an 3*2 array of type float64. By default, arrays contain 64 bit[5] floating point numbers. You can specify the particular array type by using the keyword dtype.*

```
a = np.zeros([m,n],dtype=int)
print a.dtype
int64
```

[5] On your computer, particularly if you have an older computer, `int` might denote 32 bits integers

You can also create arrays from lists of numbers:

```
a = np.array([[2,3],[3,4]])
print a
[[2 3]
 [3 4]]
```

There are many more ways to create arrays in numpy and we will get to see them as we progress in the classes.

0.2.2 Some Matrix Operations and Properties

- Product of two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ is the matrix $C = AB \in \mathbb{R}^{m\times p}$, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}.$$

Exercise 0.10 *You can multiply two matrices by looping over both indexes and multiplying the individual entries.*

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,1],[1,1]])
a_dim1, a_dim2 = a.shape
b_dim1, b_dim2 = b.shape
c = np.zeros([a_dim1,b_dim2])
for i in xrange(a_dim1):
    for j in xrange(b_dim2):
        for k in xrange(a_dim2):
            c[i,j] += a[i,k]*b[k,j]
print c
```

This is, however, cumbersome and inefficient. Numpy supports matrix multiplication with the dot function:

```
d = np.dot(a,b)
print d
```

Important note: *with numpy, you must use* `dot` *to get matrix multiplication, the expression a * b denotes element-wise multiplication.*

- Matrix multiplication is associative: $(AB)C = A(BC)$.
- Matrix multiplication is distributive: $A(B + C) = AB + AC$.
- Matrix multiplication is (generally) not commutative : $AB \neq BA$.
- Given two vectors $x, y \in \mathbb{R}^n$ the product $x^T y$, called **inner product** or **dot product**, is given by

$$x^T y \in \mathbb{R} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i.$$

```
a = np.array([1,2])
b = np.array([1,1])
np.dot(a,b)
```

- Given vectors $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$, the **outer product** $xy^T \in \mathbb{R}^{m \times n}$ is a matrix whose entries are given by $(xy^T)_{ij} = x_i y_j$,

$$xy^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}.$$

```
np.outer(a,b)
array([[1, 1],
       [2, 2]])
```

- The **identity matrix**, denoted $I \in \mathbb{R}^{n \times n}$, is a square matrix with ones on the diagonal and zeros everywhere else. That is,

$$I_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

It has the property that for all $A \in \mathbb{R}^{n \times n}$, $AI = A = IA$.

```
I = np.eye(2)
x = np.array([2.3, 3.4])

print I
print np.dot(I,x)

[[ 1.,  0.],
 [ 0.,  1.]]
[2.3, 3.4]
```

- A **diagonal matrix** is a matrix where all non-diagonal elements are 0.
- The **transpose** of a matrix results from "'flipping" the rows and columns. Given a matrix $A \in \mathbb{R}^{m \times n}$, the transpose $A^T \in \mathbb{R}^{n \times m}$ is the $n \times m$ matrix whose entries are given by $(A^T)_{ij} = A_{ji}$.

  Also, $(A^T)^T = A; \quad (AB)^T = B^T A^T; \quad (A + B)^T = A^T + B^T$

  In numpy, you can access the transpose of a matrix as the `T` attribute:

```
A = np.array([ [1, 2], [3, 4] ])
print A.T
```

- A square matrix $A \in \mathbb{R}^{n \times n}$ is **symmetric** if $A = A^T$.
- The **trace** of a square matrix $A \in \mathbb{R}^{n \times n}$ is the sum of the diagonal elements, $\text{tr}(A) = \sum_{i=1}^{n} A_{ii}$

0.2.3 Norms

The **norm** of a vector is informally the measure of the "length" of the vector. The commonly used Euclidean or ℓ_2 norm is given by

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

- More generally, the ℓ_p norm of a vector $x \in \mathbb{R}^n$, where $p \geq 1$ is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Note: ℓ_1 norm : $\|x\|_1 = \sum_{i=1}^{n} |x_i| \qquad \ell_\infty$ norm : $\|x\|_\infty = \max_i |x_i|$.

0.2.4 Linear Independence, Rank, and Orthogonal Matrices

A set of vectors $\{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^m$ is said to be **(linearly) independent** if no vector can be represented as a linear combination of the remaining vectors. Conversely, if one vector belonging to the set can be represented as a linear combination of the remaining vectors, then the vectors are said to be **linearly dependent**. That is, if

$$x_j = \sum_{i \neq j} \alpha_i x_i$$

for some $j \in \{1, \ldots, n\}$ and some scalar values $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n \in \mathbb{R}$.

- The **rank** of a matrix is the number of linearly independent columns, which is always equal to the number of linearly independent rows.
- For $A \in R^{m \times n}$, $\text{rank}(A) \leq \min(m, n)$. If $\text{rank}(A) = \min(m, n)$, then A is said to be **full rank**.
- For $A \in R^{m \times n}$, $\text{rank}(A) = \text{rank}(A^T)$.
- For $A \in R^{m \times n}$, $B \in R^{n \times p}$, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$.
- For $A, B \in R^{m \times n}$, $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$.
- Two vectors $x, y \in \mathbb{R}^n$ are **orthogonal** if $x^T y = 0$. A square matrix $U \in \mathbb{R}^{n \times n}$ is orthogonal if all its columns are orthogonal to each other and are normalized ($\|x\|_2 = 1$), It follows that

$$U^T U = I = U U^T.$$

Bibliography

Pérez, F. and Granger, B. E. (2007). IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29.