Chapter 1

Basic Tutorials

1.1 Today's assignment

In this lab we will introduce several fundamental concepts needed further ahead. We start with an 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. The goal of this lab is to give you the basic familarity of tools you will need for the homeworks and the final project.

Please feel free to skip any part of this lab if you are familiar with the tool.

This lab guide is taken from Lisbon Machine Learning School `http://lxmls.it.pt/2018/LxMLS_guide_2018.pdf`

1.2 Manually Installing the Tools in your own Computer

1.2.1 Installing Python from Scratch with Anaconda

If you are new to Python the best option right now is the Anaconda platform. You can find installers for Windows, Linux and OSX platforms here

```
https://www.anaconda.com/download/
https://anaconda.org/pytorch/pytorch
```

Use python 3. Do not use python 2.

Important: As the labs progress you will need an IDE, or at least a good editor and knowledge of pdb/ipdb. This will not be obvious the first days since we will be seeing simpler examples.

Easy IDEs to work with Python are PyCharm and Visual Studio Code, but feel free to use the software you feel more comfortable with. PyCharm and other well known IDEs like Spyder are provided with the Anaconda installation.

Aside of an IDE, you will need an interactive command line to run commands. This is very useful to explore variables and functions and quickly debug the exercises. For the most complex exercises you will still need an IDE to modify particular segments of the provided code. As interactive command line we recommend the Jupyter notebook. This also comes installed with Anaconda and is part of the pip-installed packages. The Jupyter notebook is described in the next section. In case you run into problems or you feel uncomfortable with the Jupyter notebook you can use the simpler iPython command line.

1.2.2 Jupyter Notebook

Jupyter is a good choice for writing Python code. It is an interactive computational environment for data science and scientific computing, where you can combine code execution, rich text, mathematics, plots and rich media. The Jupyter Notebook is a web application that allows you to create and share documents, which contains live code, equations, visualizations and explanatory text. It is very popular in the areas of data cleaning and transformation, numerical simulation, statistical modeling, machine learning and so on. It supports more than 40 programming languages, including all those popular ones used in Data Science such as Python, R, and Scala. It can also produce many different types of output such as images, videos, LaTex and JavaScript. More over with its interactive widgets, you can manipulate and visualize data in real time.
The main features and advantages using the Jupyter Notebook are the following:

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

The basic commands you should know are

Esc	Enter command mode
Enter	Enter edit mode
up/down	Change between cells
Ctrl + Enter	Runs code on selected cell
Shift + Enter	Runs code on selected cell, jumps to next cell
restart button	Deletes all variables (useful for troubleshooting)

Table 1.1: Basic Jupyter commands

A more detailed user guide can be found here:

`http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/index.html`

1.3 Python

1.3.1 Python Basics

Pre-requisites

At this point you should have installed the needed packages. You need also to feel comfortable with an IDE to edit code and an interactive command line. See previous sections for the details. Create a folder and change the current directory into that folder.
from there, start your interactive command line of choosing, e.g.,

```
jupyter-notebook
```

and proceed with the following sections.

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 the IDE of your choice, e.g., PyCharm:

```python
print('Hello World!')
```

After saving and closing the file, you can run your code by using the run functionality in your IDE. If you wish to run from a command line instead do

```
python yourfile.py
```

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

In the interactive command line – Start your preferred interactive command line, e.g., Jupyter-notebook. There, you can run Python code by simply writing it and pressing enter (ctr+enter in Jupyter).

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

However, 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 directly in the command line and, e.g., inspect variables. Larger sections of code can be stored and run from files.

Help and Documentation

There are several ways to get help on Jupyter:

- 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('if')` gets the online documentation for the `if` keyword;
- `help()`, enters the help system.
- When at the help system, type `q` to exit.

1.3.2 Python by Example

Basic Math Operations

Python supports all basic arithmetic operations, including exponentiation. For example, the following code:

```
print(3 + 5)
print(3 - 5)
print(3 * 5)
print(3 / 5)
print(3 ** 5)
```

will produce the following output in Python 2:

```
8
-2
15
0
243
```

and the following output in Python 3:

```
8
-2
15
0.6
243
```

Important: Notice that in Python 2 division is always considered as integer division, hence the result being 0 on the example above. To force a floating point division in Python 2 you can force one of the operands to be a floating point number:

```
print(3 / 5.0)
0.6
```

For Python 3, the division is considered float point, so the operation (3 / 5) or (3 / 5.0) is always 0.6.
Also, notice that the symbol `**` is used as exponentation operator, unlike other major languages which use the symbol `^`. In fact, the `^` symbol has a different meaning in Python (bitwise XOR) so, in the beginning, be sure to double-check your code if it uses exponentiation and it is giving unexpected results.

Data Structures

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

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

A string should be surrounded by either apostrophes (′) or quotes (“). 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 $j-1$, inclusively.

Exercise 1.1 *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 over 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 in Jupyter. Alternatively you can write this into your `yourfile.py` file and run it. Do you 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 of a statement, e.g., `while` or `for`. 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 1.2 *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 to 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. The recommended[1] way is to use 4 spaces for each indentation level.

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!')
```

[1]The PEP8 document (`www.python.org/dev/peps/pep-0008`) is the official coding style guide for the Python language.

You can write this command into Jupyter directly or write it into a file which you then run in Jupyter. 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 1.3 *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 Jupyter. 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, there will be situations in which we will use and extend modules that involve 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. For this to be possible is very important that you learn to use an IDE.

An alternative to IDEs, that can also be useful for quick debugging in Jupyter, is the pdb module. This will 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. The ipdb is an improved version of pdb that has to be installed separately. It provides additional functionalities like larger context windows, variable auto complete and colors. Unfortunately ipdb has some compatibility problems with Jupyter. We therefore recommend to use ipdb only in spartan configurations such as vim+ipdb as IDE.
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 pdb; pdb.set_trace()
        print('Good evening!')
```

Load the new definition of the function by writing this code in a file or a Jupyter cell 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[2], but we provide here a short table with the most useful ones:

(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
b(reak) n, condition	Sets a conditional 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)
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 1.2: Basic pdb/ipdb commands, parentheses indicates abbreviation

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

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

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

```
pdb> 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.

1.3.3 Exceptions

Occasionally, 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
```

[2]The complete list can be found at `http://docs.python.org/library/pdb.html`

A *ZeroDivisionError* exception was raised, and no output was returned. Exceptions can also be forced to occur by the programmer, with customized error messages [3].

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

Exercise 1.4 *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(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 exception named in the *except* keyword, the except clause is executed; otherwise, the exception is raised and execution is aborted (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 Jupyter 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.

Organizing your Code with your own modules

Creating you own modules is extremely simple. you can for example create the file in your work directory

`my_tools.py`

and store there the following code

```
def my_print(input):
    print(input)
```

From Jupyter you can now import and use this tool as

```
import my_tools
my_tools.my_print("This works!")
```

[3]For a complete list of built-in exceptions, see `http://docs.python.org/3/library/exceptions.html`

Important: When you modify a module, you need to reload the notebook page for the changes to take effect. Autoreload is set by default in the schools notebooks.
for the latter. Other ways of importing one or all the tools from a module are

```
from my_tools import my_print   # my_print directly accesible in code
from my_tools import *          # will make all functions in my_tools accessible
```

However, this makes reloading the module more complicated. You can also store tools ind different folders. For example, if you store the previous example in the folder

`day0_tools`

and store inside an empty file called

`__init__.py`

then the following import will work

```
import day0_tools.my_tools
```

1.3.4 Matplotlib – Plotting in Python

Matplotlib[4] 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).

```
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 1.5 *Try running the following on Jupyter, 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)
```

[4]http://matplotlib.org/

```
# 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()
```

1.3.5 Numpy – Scientific Computing with Python

Numpy[5] 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 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.

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)
```

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

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 1.6 *Run the above example and lookup the* `ptp` *function/method (use the* `?` *functionality in Jupyter).*

Exercise 1.7 *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?*