Data analysis in Python, the literate way

NOTE: Code in this document has been tested with Python 2.7.7. However, it may work also with previous and newer versions.

Tabla de contenido

1. IPython and its distributions

IPython

The IPython interpreter is an enhanced Python shell. This means it features a bunch of additional useful features for numerical/scientific computation.

While the most obvious features for the end user are related to the user interface and the use of notebooks, IPython is actually a distributed computational environment that allows for lightweight parallelization. For example, if you need to process big arrays or matrices in a computing intensive task, you can distribute that processing in your own cluster, or a cluster in Amazon or MS Azure.

KNOW MORE: You can get some ideas on how different parallel computing frameworks are abstracted out in IPython here: http://ipython.org/ipython-doc/dev/parallel/parallel_intro.html (advanced)

SciPy

SciPy is a large collection of scientific, machine learning and engineering libraries. You can think of it as a large library where you can find any method, tool or utility you need for data analysis.
The core packages are described here: http://www.scipy.org/about.html

The Anaconda Distribution

Anaconda is a distribution of IPython and ScyPy made by Continuum Analytics. You can think on it as a distribution of different tools that together make a workbench for data analysis.

If you are curious of what is included there you can see the list here:
http://docs.continuum.io/anaconda/pkg-docs.html

2. The IPython shell

IPython does not provide any scientific or data analysis tools itself, but **promotes an "execute-explore" workflow** that is well suited to data science work. It also features a lightweight, fast parallel computing engine that can be used for more computing-intensive tasks.

When opening the IPython shell, you will see some differences from normal Python.

```
[vagrant@localhost ~]$ ipython
Python 2.7.7 |Anaconda 2.0.1 (64-bit)| (default, Jun  2
2014, 12:34:02)
Type "copyright", "credits" or "license" for more
information.

IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and
https://binstar.org
?         -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for
extra details.

In [1]:
```

In principle, the shell works in a similar way as the "normal" Python interpreter:

```
In [3]: a = range(1, 20, 5)

In [4]: a
Out[4]: [1, 6, 11, 16]

In [5]:
```

One of the important features Tab completion. This means that if you press <Tab> in the prompt it will try to complete what you are writing.

For example:

```
In [1]: import pandas.<Tab>
pandas._sparse      pandas.io           pandas.sparse
pandas._testing     pandas.json         pandas.stats
pandas.algos        pandas.lib          pandas.tests
pandas.compat       pandas.msgpack      pandas.tools
pandas.computation  pandas.np           pandas.tseries
pandas.core         pandas.offsets      pandas.tslib
```

```
pandas.datetools    pandas.pandas       pandas.util
pandas.hashtable    pandas.parser       pandas.version
pandas.index        pandas.rpy
pandas.info         pandas.sandbox
```

This works for modules, but also for variables, functions and for any input in general.

Getting help

You can get info on any object using the question mark. For example:

```
In [6]: a = range(1,20,6)

In [7]: a?
Type:        list
String form: [1, 7, 13, 19]
Length:      4
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's
items
```

This is called **introspection**. With two question marks you will get more information if available (normally, the source code).

It also works with other objects as functions:

```
In [12]: range??
Type:        builtin_function_or_method
String form: <built-in function range>
Namespace:   Python builtin
Docstring:
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to
0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is
omitted!
These are exactly the valid indices for a list of 4 elements.
```

You can also use the Python help system for a similar purpose.

```
In [2]: help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should
definitely check out the tutorial on the Internet at
http://docs.python.org/2.7/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help
on writing Python programs and using Python modules.  To
quit this help utility and return to the interpreter, just
type "quit".

To get a list of available modules, keywords, or topics,
type "modules","keywords", or "topics".  Each module also
comes with a one-line summary of what it does; to list the
modules whose summaries contain a given Word such as "spam",
type "modules spam".

help> random
Help on module random:

NAME
    random - Random variable generators.
FILE
    /home/vagrant/anaconda/lib/python2.7/random.py
MODULE DOCS
    http://docs.python.org/library/random
DESCRIPTION
        integers
        --------
               uniform within range
        sequences
        ---------
               pick random element
               pick random sample
               generate random permutation
…
```

Using magics

Magics commands are used to facilitate common tasks in IPython. They are written starting with a percent symbol. For example, you can see the variables defined in the current environment as follows:

```
In [7]: a = 5.2

In [8]: b = range(5)

In [9]: %whos
Variable   Type     Data/Info
----------------------------
a          float    5.2
b          list     n=5
```

There are many magic commands and you can see them all using lsmagic:

```
In [10]: %lsmagic
Out[10]:
```

```
Available line magics:
%alias  %alias_magic  %autocall  %autoindent  %automagic
%bookmark  %cat  %cd  %clear  %colors  %config  %cp  %cpaste
%debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui
%hist  %history  %install_default_config  %install_ext
%install_profiles  %killbgscripts  %ldir  %less  %lf  %lk
%ll  %load  %load_ext  %loadpy  %logoff  %logon  %logstart
%logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic
%man  %matplotlib  %mkdir  %more  %mv  %notebook  %page
%paste  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo
%pinfo2  %popd  %pprint  %precision  %profile  %prun
%psearch  %psource  %pushd  %pwd  %pycat  %pylab  %quickref
%recall  %rehashx  %reload_ext  %rep  %rerun  %reset
%reset_selective  %rm  %rmdir  %run  %save  %sc  %store  %sx
%system  %tb  %time  %timeit  %unalias  %unload_ext  %who
%who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file
%%html  %%javascript  %%latex  %%perl  %%prun  %%pypy
%%python  %%python2  %%python3  %%ruby  %%script  %%sh
%%svg  %%sx  %%system  %%time  %%timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics
```

You can see from the list that there are many magic commands that implement shell commands. For example:

```
In [16]: %pwd
Out[16]: u'/home/vagrant'

In [17]: %ls
anaconda/  msicilia/
```

You can get a short explanation of all the magics with %quickref.

NOTE: If you want to use the value of a IPython variable in one of the shell commands, you have to use the $ sign. For example: %mkdir $myvar

You can run scripts from inside IPython with the run magic. The global variables defined in the script are kept in the engine associated to the IPython session. For example for the following test.py file:

```
In [6]: %cat test.py
a=5
print a

In [7]: %run test.py
5

In [8]: a = a +1
```

```
In [9]: print a
6
```

Note that running a script with %run clears the memory, if you want to retain the globals you already have in your session, use `%run` with the `–i` modifier.

NOTE: You can create bookmarks with the %bookmark magic to put labels to concrete folders for example. This helps in avoiding navigating to different folders using `cd` commands.

Clearing the memory

It is important to notice that everything we do in the IPython session is stored in the memory of the session. In some cases, we would like to start clean, and we can use the following for that purpose:

```
In [10]: %reset
Once deleted, variables cannot be recovered. Proceed
(y/[n])? y
```

The IPython history

In a normal Python shell, you can use the arrow keys to get the last commands you typed. In IPython, that history is accessible across sessions.

You can also have access to previous inputs and outputs in the history, for example:

```
In [10]: a = 5

In [11]: a = a+5

In [12]: In[10]
Out[12]: u'a = 5'

In [13]: 34*23
Out[13]: 782

In [14]: print a
10

In [15]: b = _

In [16]: print b
782
```

You can also do more with the history or even save it all to a file with the %history magic.

2. Literate programming with notebooks

From Wikipedia:

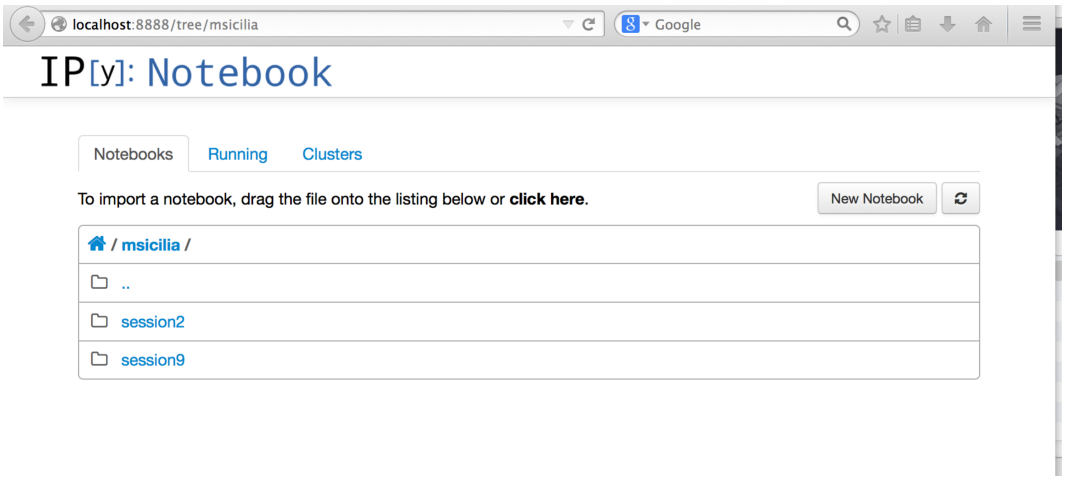
> **Literate programming** is an approach to programming introduced by Donald Knuth in which a program is given as an explanation of the program logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which a compilable source code can be generated.

This is the idea behind IPython Notebooks. An IPython Notebook is a Web-based interactive computational environment where you can mix code execution, text, mathematics, plots and rich media into a single document.

You can start the notebook Web interface simply by using the following command.

```
$ ipython notebook
```

You will see that a browser pops up with the following interface, and your terminal window keeps showing messages about what is going on behind the scenes.




```
[vagrant@localhost ~]$ ipython notebook
2014-10-28 14:45:08.818 [NotebookApp] Using existing profile
dir: u'/home/vagrant/.ipython/profile_default'
2014-10-28 14:45:08.822 [NotebookApp] Using MathJax from
CDN: https://cdn.mathjax.org/mathjax/latest/MathJax.js
2014-10-28 14:45:08.833 [NotebookApp] Serving notebooks from
local directory: /home/vagrant
2014-10-28 14:45:08.833 [NotebookApp] 0 active kernels
```

```
2014-10-28 14:45:08.833 [NotebookApp] The IPython Notebook
is running at: http://0.0.0.0:8888/
2014-10-28 14:45:08.833 [NotebookApp] Use Control-C to stop
this server and shut down all kernels (twice to skip
confirmation).
INFO:tornado.access:302 GET / (10.0.2.2) 1.35ms
WARNING:tornado.access:404 GET /favicon.ico (10.0.2.2)
0.99ms referer=None
WARNING:tornado.access:404 GET /favicon.ico (10.0.2.2)
0.51ms referer=None
```

The page in the previous screenshot is the notebook dashboard; it lists all notebooks in the folder where we launched ipython notebook from. An IPython notebook file has a .ipynb extension; it is a text file containing structured data in JSON. You can see there three tabs:

- **Notebooks**: It is actually a web interface to the file system, giving you the option to create new notebooks in the folder you are positioned.
- **Running**: It shows the notebooks that are running and allows you to shutdown them. Each notebook has at least one kernel associated, i.e. a process for executing the Python code we write and send.
- **Cluster**: this shows the clusters of IPython engines (more on this in future sessions). This is only useful if you are doing parallel programming with IPython.

NOTE: We will be using Notebooks executing its kernels inside our own computers. But this not needs to be the setup. You could be executing the kernel in remote. For example, the NotebookCloud app allows you to execute them in your Amazon Web Services account: https://notebookcloud.appspot.com/docs

Once you create a Notebook, its kernel will be running and you enter the interactive environment.

Editing the notebook

When editing the Notebook, we have to think on it as a series of **Cells**. Cells can contain fragments of Python code, but they can also contain text. The default type of cell is “code” but you can change it with the dropdown in the toolbar or with the Cell|Cell Type menu.

You can use different level of text headings, and also “markdown”. Markdown is a sort of concise formatting notation, go to Help|Markdown to learn a bit about it. Markdown includes also a subset of HTML. You can also use Latex to display beautiful formulae.

The Edit menu allows you to insert cells in various ways, deleting them, doing cut, copy and paste of cells. It also allows for merging and splitting cells or moving them.

Executing code

You can execute code in a cell with several key combinations, using the toolbars or the Cell menu. For example, <Ctrl>+<Enter> executes the code and includes the input. With <Shift>+<Enter>, the same happens but a new cell is added below.

Take into account that the variables and imports get executed and persists till you shutdown the kernel. You could do this explicitly (or restart it) from the Kernel menu.

Integration with plotting

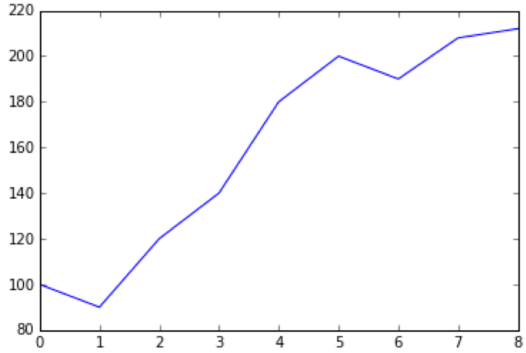
The matplotlib package allows for including plots inside IPython cells (by default it opens a new window with the plot). For doing so, we need to execute a magic and then the plot appears inside the HTML:

```
In [48]: monthly_data_2013 = [100, 90, 120, 140, 180, 200, 190, 208, 212]
```

A look at the data

The first thing to do with data is taking a look at it. In our case, we will start by plotting it.

```
In [49]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.plot(monthly_data_2013)
Out[49]: [<matplotlib.lines.Line2D at 0x7f124b439fd0>]
```



Closing, copying and exporting notebooks

In the File menu, you can create new notebooks, rename or create copies of the current one, and also export the Notebook to other formats. For example, you can export them to HTML. If you are working against a remote machine, you can download it in the IPython format to have it locally.

NOTE: You can share an HTML version of your notebook in the NBViewer site: http://nbviewer.ipython.org/

4. Data science in a nutshell

A naïve forecasting model

See the "Starting Data Science!" notebook.

5. Timing and profiling

Timing and profiling code

When working with Big Data, the performance of each function and sentence in some cases matters. This is why a very common task in the data scientist workflow is that of looking at the time it takes to execute fragments of code.

Timing code can be done by using the built-in Python `time` module that allows capturing the current time of the system. However, in IPython we have two magic functions to ease that task. The first one allows us to see the time taken by a single sentence or expression. Let's look at an example.

```
In [1]: biglist = range(1000000)

In [2]: bigset = set(biglist)

In [6]: %time 500000 in biglist
CPU times: user 9 ms, sys: 0 ns, total: 9 ms
Wall time: 9.63 ms
Out[6]: True

In [7]: %time 500000 in biglist
CPU times: user 9 ms, sys: 0 ns, total: 9 ms
Wall time: 9.68 ms
Out[7]: True
```

As it can be appreciated, two executions may lead to different timing, as there are many things going on in our computer that may cause that difference. This is why usually the idea is averaging multiple executions of the same fragment of code. We can do it as illustrated in the following example.

```
In [3]: %timeit 500000 in biglist
100 loops, best of 3: 9.02 ms per loop

In [4]: %timeit 500000 in bigset
10000000 loops, best of 3: 65.1 ns per loop

In [5]: %timeit -n 100 500000 in bigset
100 loops, best of 3: 71.5 ns per loop
```

Important things to notice:

- The magic "decides" the number of executions that fit. Basically, when the divergences in many executions are small, as with the list above, it does not continue.
- You can tell the number of execution with the –n parameter.
- This is a nice example of a apparently irrelevant decision on the data type (list versus set) that makes a big difference in performance!

KNOW MORE: If you want to understand why sets for existence operations are much better performing than lists, you need to understand how they are implemented internally. They use hash tables internally, you can get a basic understanding of hashing from here: http://www.youtube.com/watch?v=AArXvYMTCOM

With this we know the global execution time, but what happens if the performance is not good in a complex program? Usually in large programs there are many function calls chained with each other, and we need to know exactly where the bottleneck is. This is called **profiling**. Let's see an example. A *profile* is no other thing that a set of statistics that describes how often and for how long various parts of the program executed.

Python provides the `cProfile` module to do that job. However, IPython provides some convenient magics to do the job. Let's consider the following example:

A glimpse at debugging

You can enter debug mode in IPython using the debug magic. This is especially useful just after an exception has occurred. For example:

```
In [14]: li = range(30)

In [15]: a = 0

In [16]: for i in range(31):
    a = a + li[i]
   ....:
```

```
---------------------------------------------------------------
---------------
IndexError                                Traceback (most
recent call last)
<ipython-input-16-5a080aad4be3> in <module>()
      1 for i in range(31):
----> 2     a = a + li[i]
      3

IndexError: list index out of range
```

```
In [18]: %debug
> <ipython-input-16-5a080aad4be3>(2)<module>()
      1 for i in range(31):
----> 2     a = a + li[i]
      3

ipdb> i
30
```

KNOW MORE: Debugging is a skill that is hard to acquire. You can get a glimpse at all the pdb functionality for example in this PyOhaio 2014 talk "So you think you can pdb": http://pyvideo.org/video/2867/so-you-think-you-can-pdb Be aware that it requires an understanding of the stack trace, i.e. the record of the Python functions that are active at a certain point of execution.