OPENCOURSEWARE
ADVANCED PROGRAMMING STATISTICS FOR DATA SCIENCE

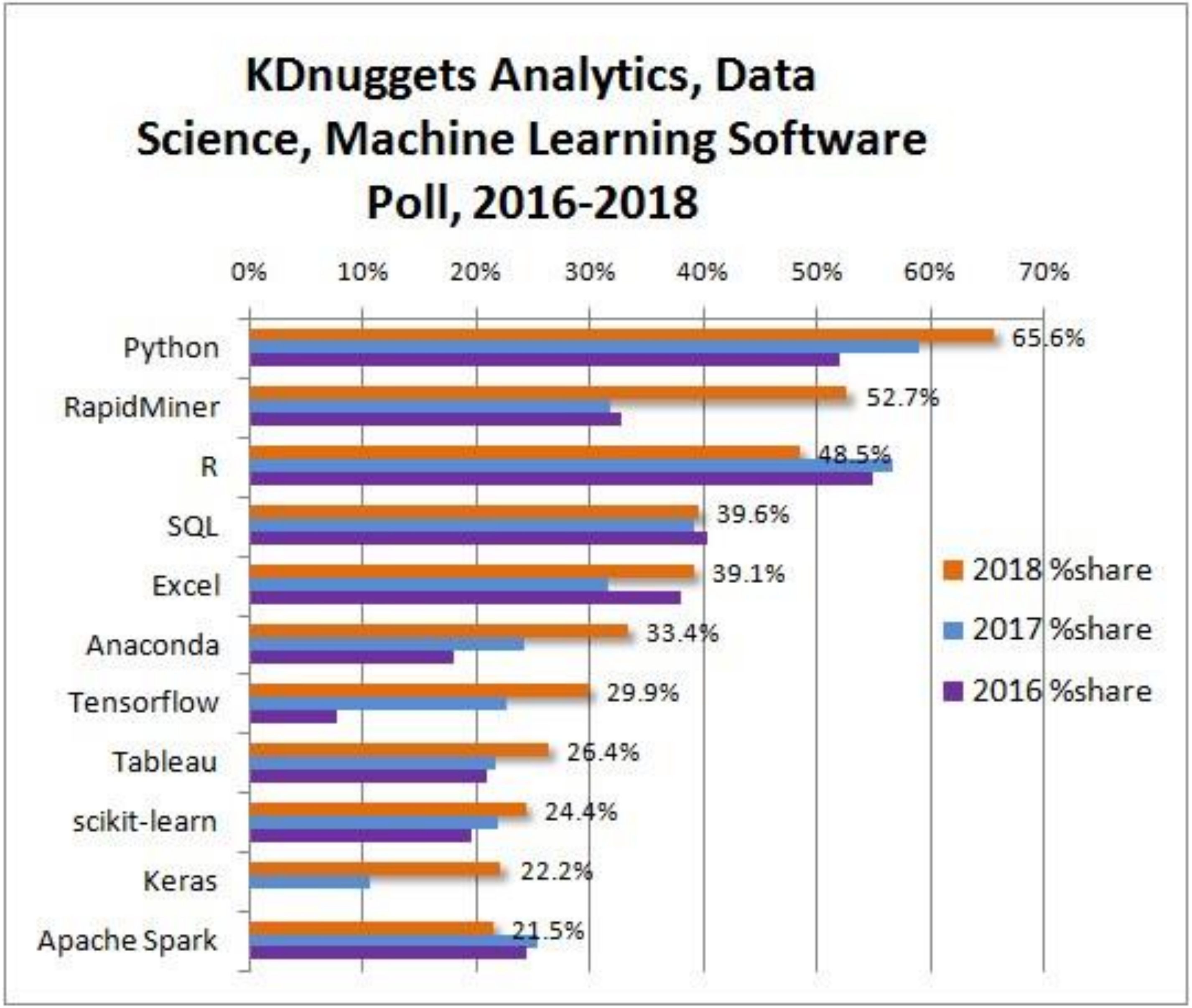
Ricardo Aler

The Python Programming Language

What is Python?

- General-purpose, high-level programming language
- Code is very readable
- Includes different ways of programming:
  - Object-oriented
  - Imperative
  - Functional programming
- Python 2.x (2.7) vs. Python 3.x (3.7)

Languages for data analysis poll



Why Python?

- Many scientific and machine learning packages: NumPy (numeric matrices), SciPy, Pandas (dataframes), Statsmodels (statistics), scikit-learn (machine learning)
- Nice interface for Spark (pyspark)
  - R's interface is not so well developed yet (sparkR, sparklyr)
- Commonly used in Deep Learning (TensorFlow, Keras, Pytorch, …)

Python versions

- Python 2.x (2.7):
  - Old version, but many packages still use it
  - It’s not going to be updated
- Python 3.x (3.7): new versión
  - But only a few differences with 2.x
  - This course we will use 3.7

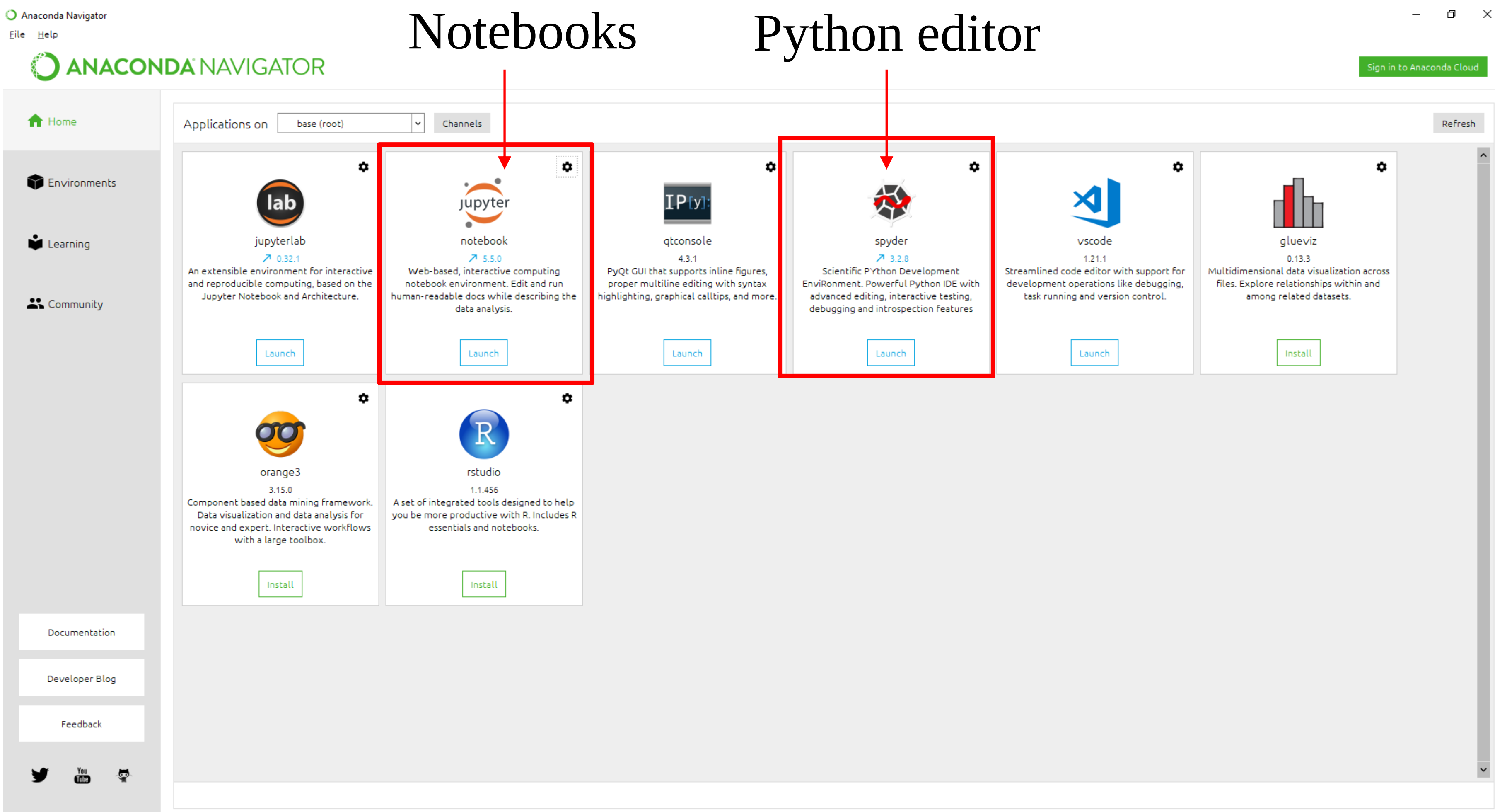
ANACONDA

- Free Python distribution. It includes over 300 of the most popular Python packages for science, math, engineering, data analysis.

Install from: https://www.anaconda.com/download/

Remember to select Python 3.7!!

Anaconda ecosystem

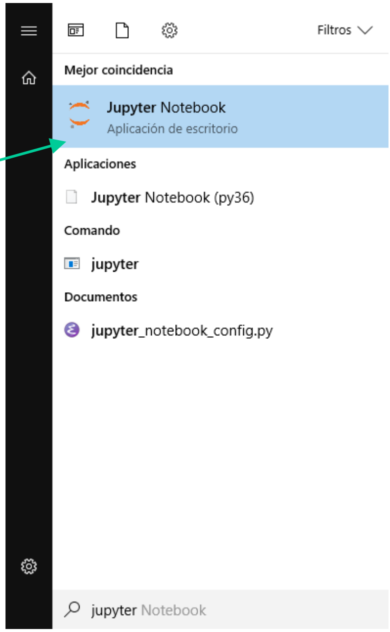


Interactive vs. Scripts

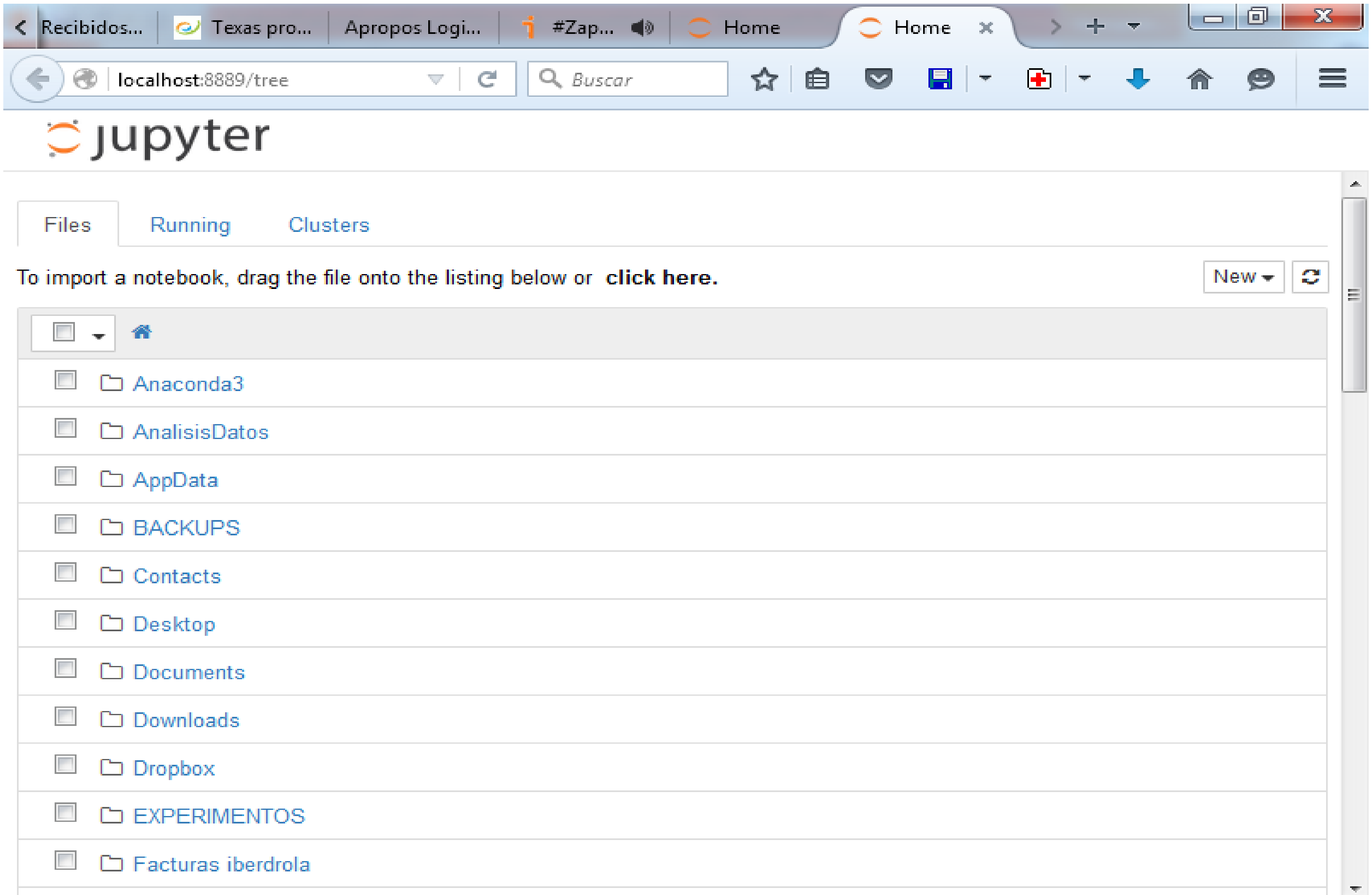
- Interactive: typing Python commands in the console (or the notebook) and obtaining an answer

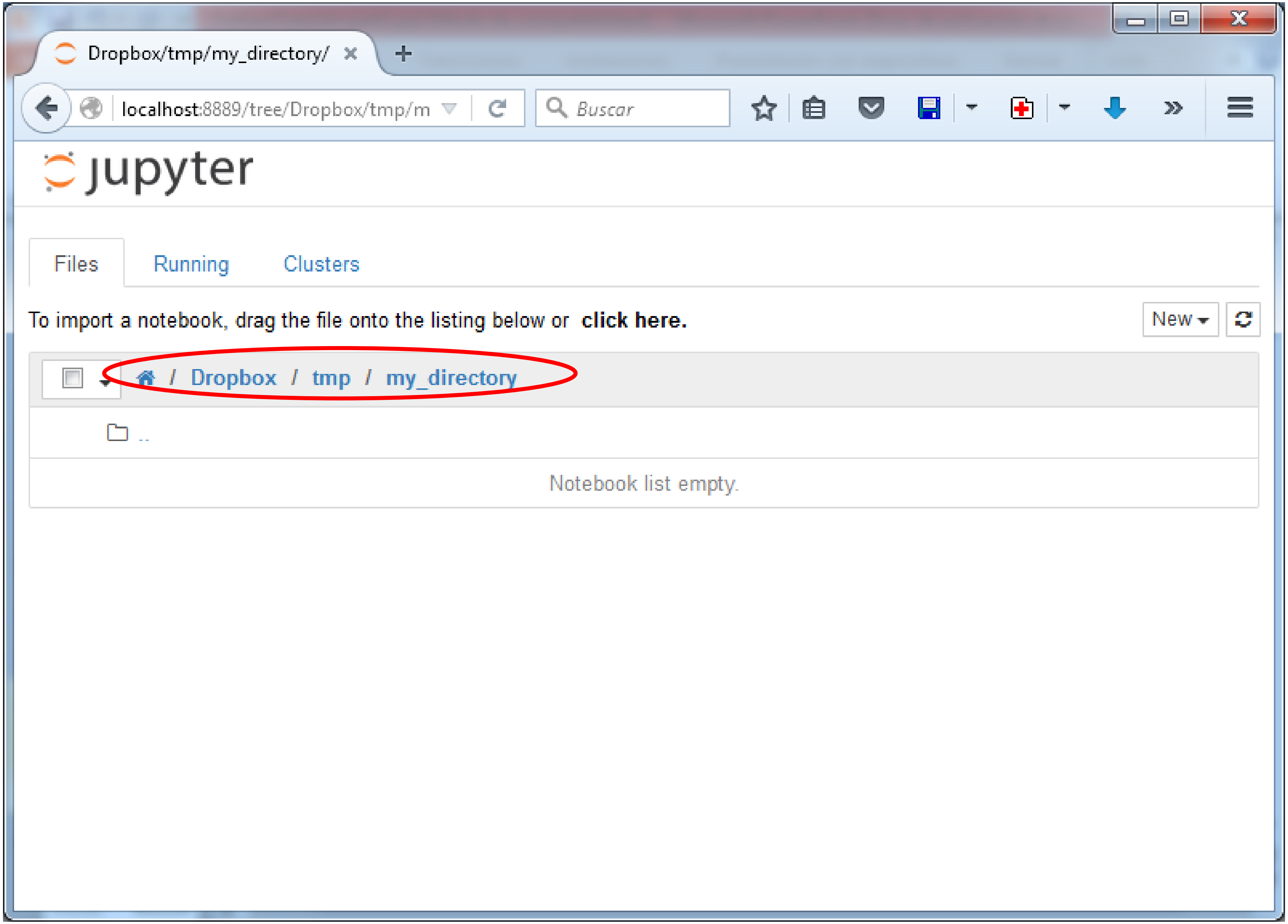
```
>>> 'hello world!'
'hello world!'
```

- Script:
  - A program is created using a text editor (for instance, with *spyder*)
  - Or using the Jupyter notebook

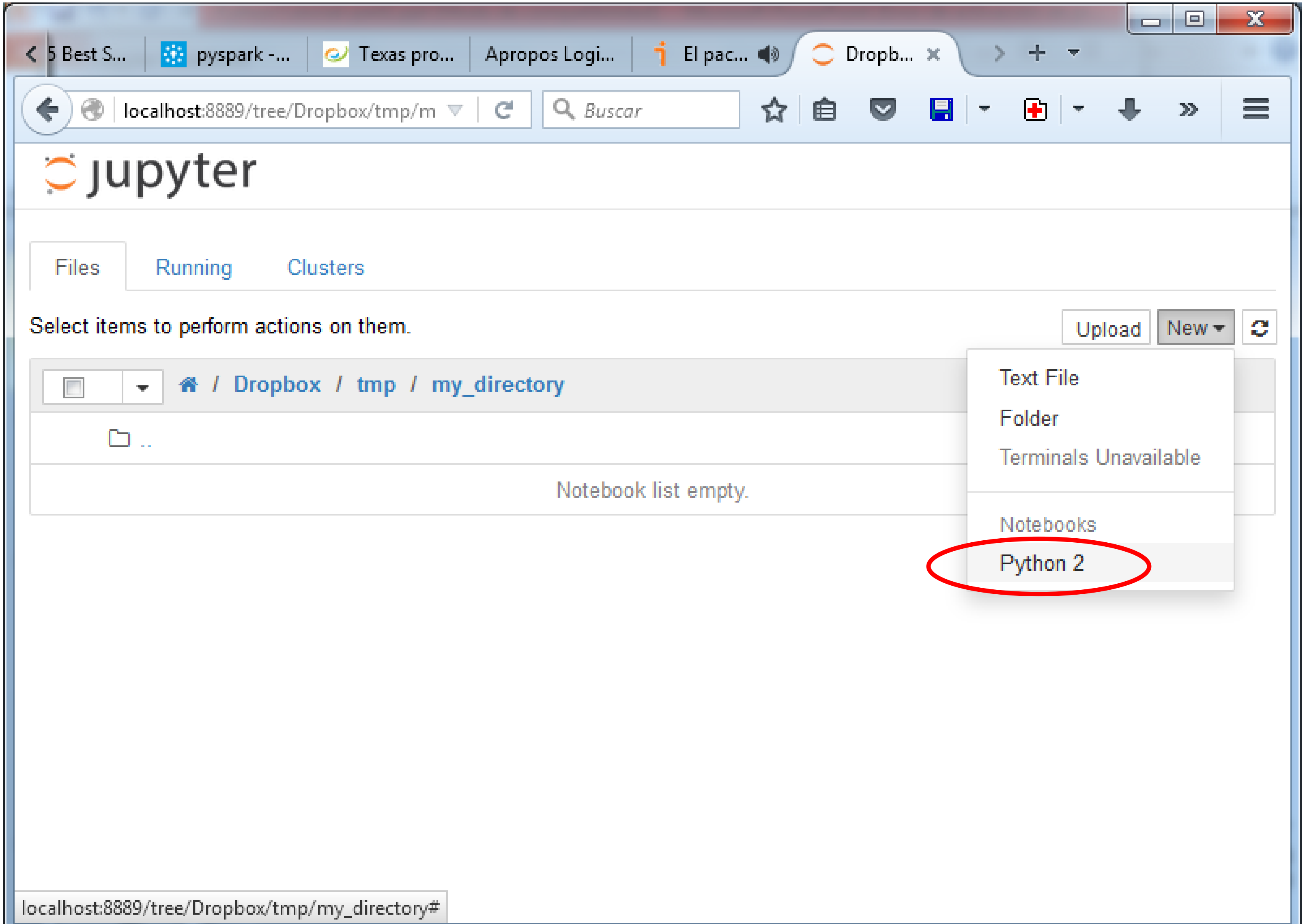


- A new tab will open in your default browser
- Now, you have to go to your directory

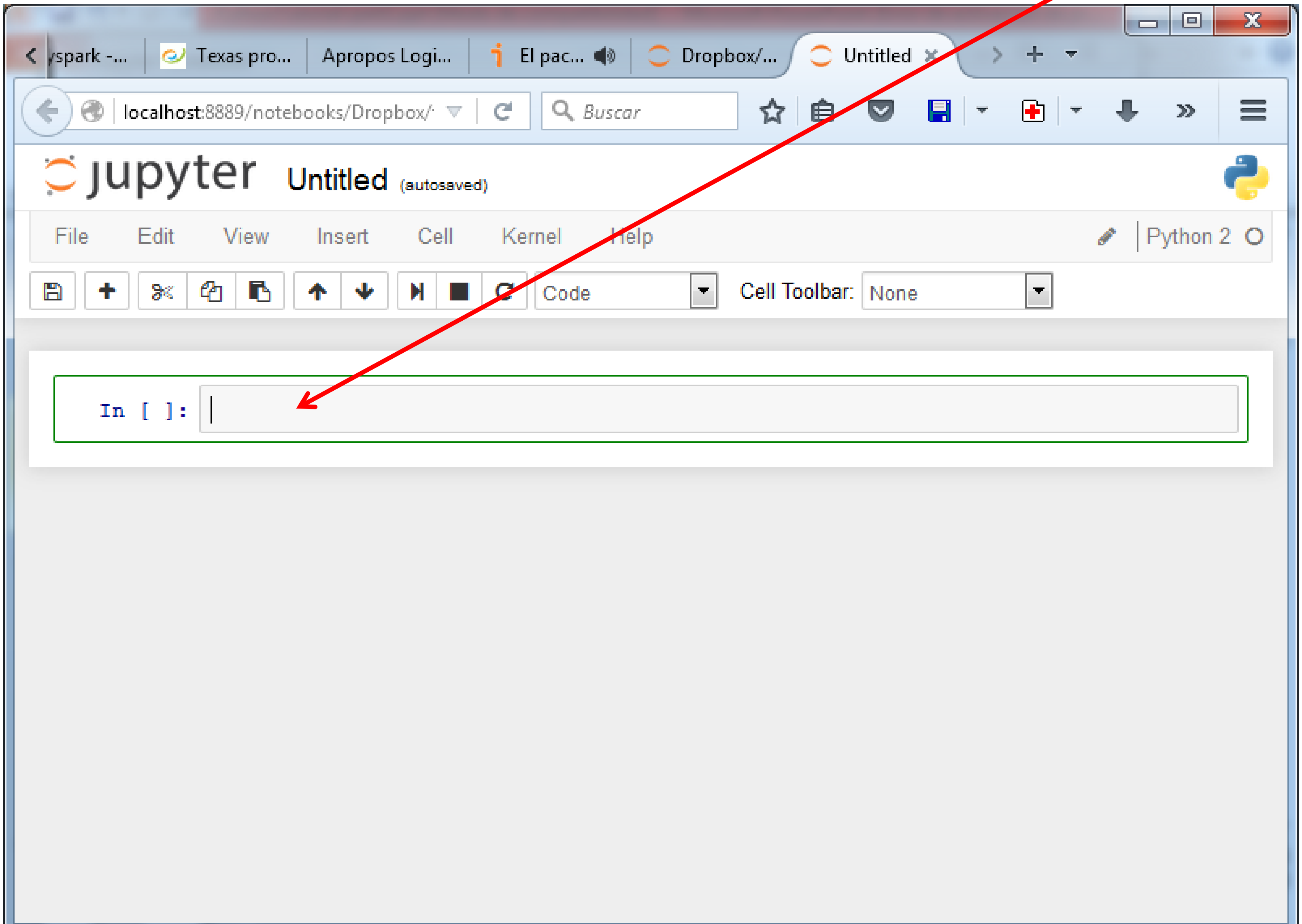



Dropbox/tmp/my_directory/
localhost:8889/tree/Dropbox/tmp/m
Buscar
jupyter
Files
Running
Clusters
To import a notebook, drag the file onto the listing below or click here.
New
/ Dropbox / tmp / my_directory
..
Notebook list empty.

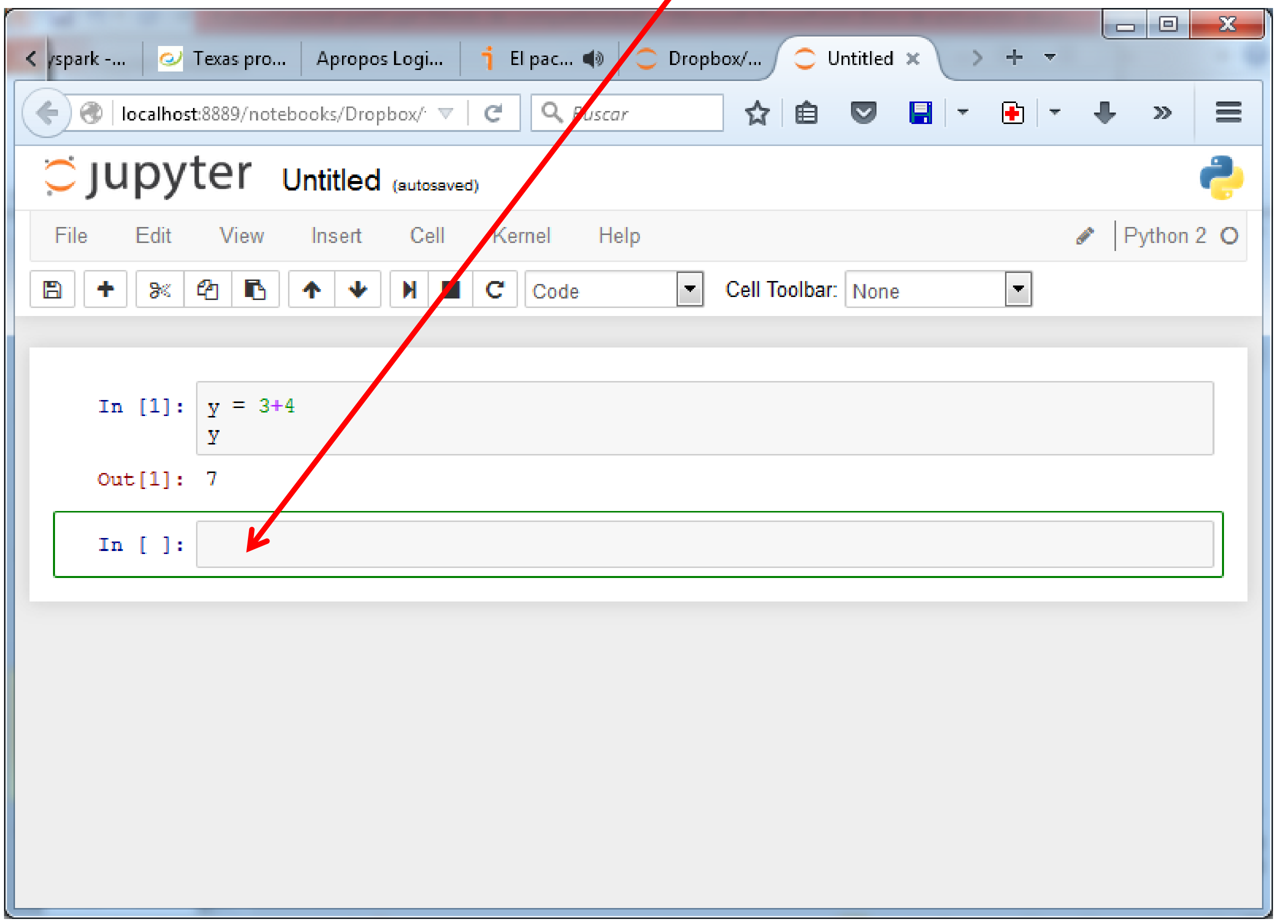
- Start a Python 2 notebook



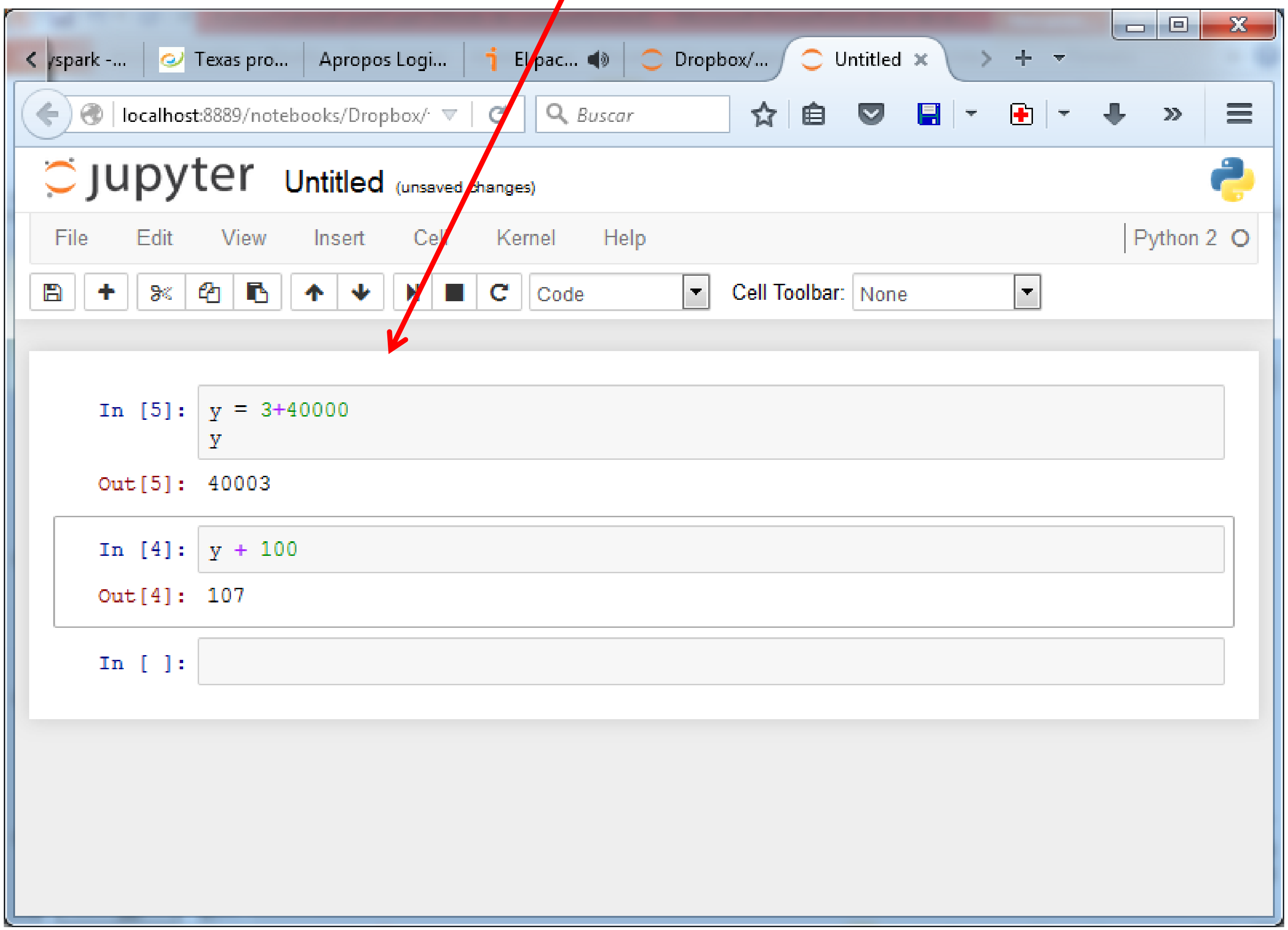
- You can type python commands in the cell



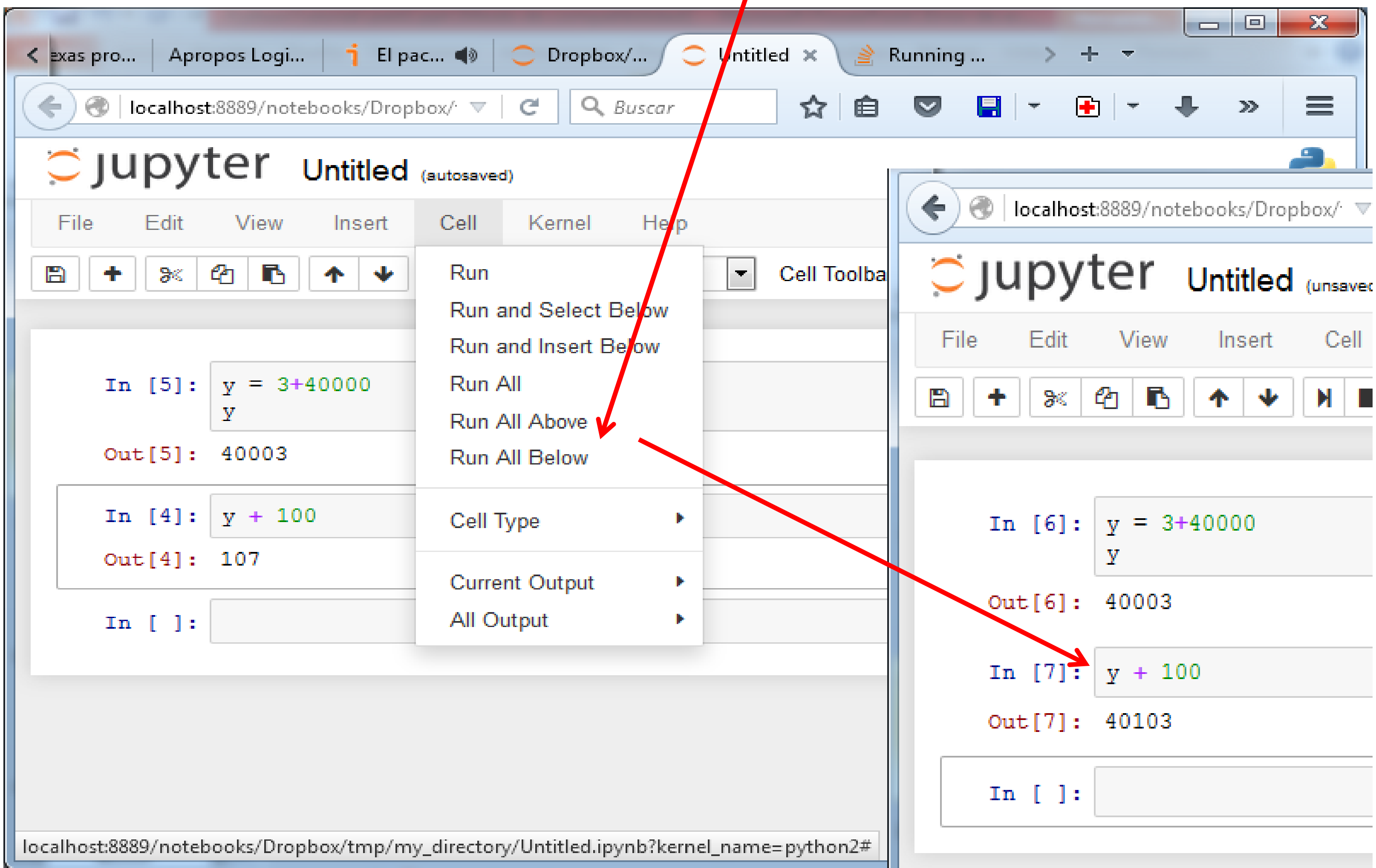
- Important:
  - “Enter” changes to a new line WITHIN the cell
  - In order to execute the commands in the cell, you have to type shift+enter
  - Once you type shift+enter, a new cell is created. You can type new commands



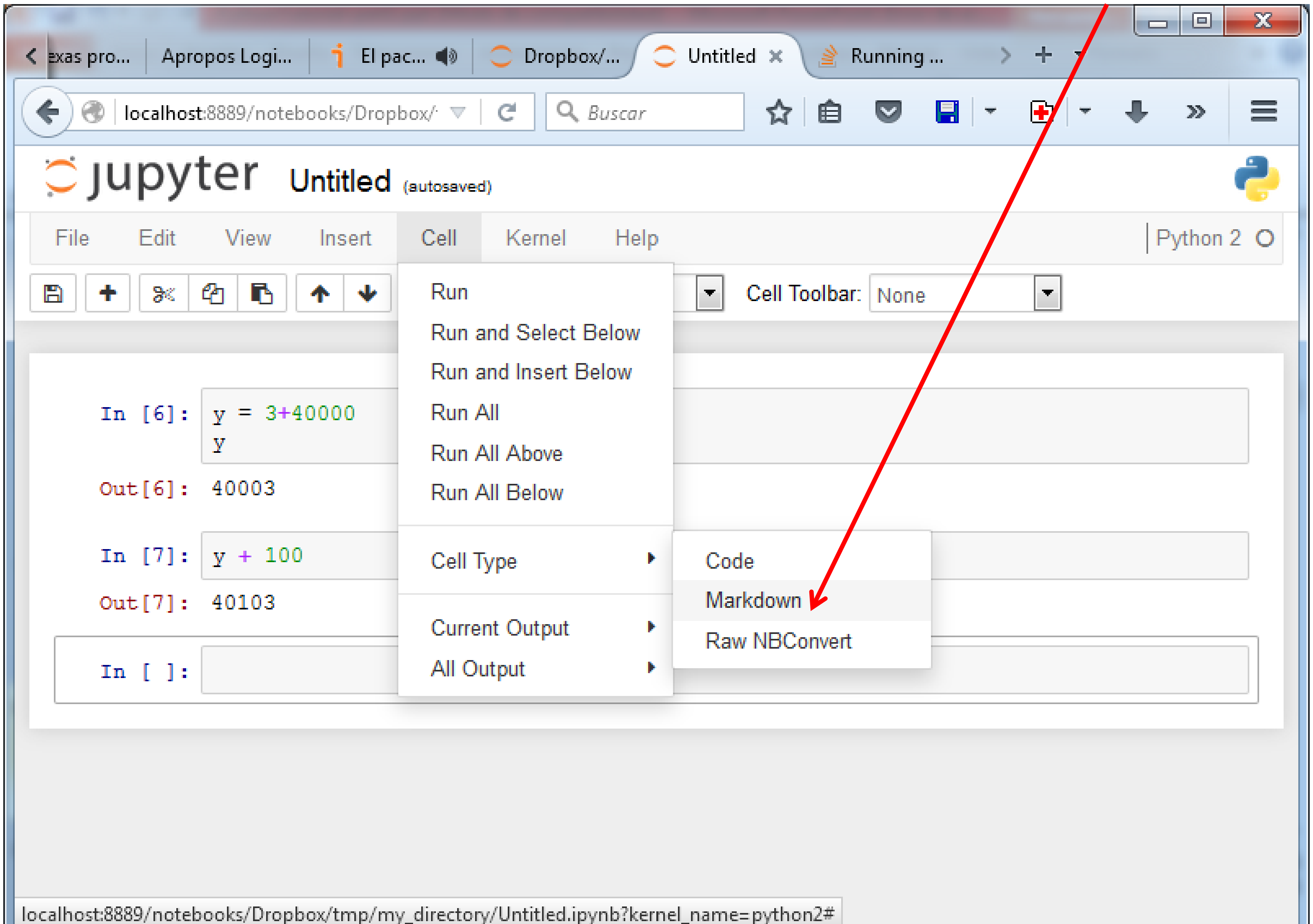
- You can return to a previous cell and change it. You need to re-execute it with shift+enter (or ctrl+enter)



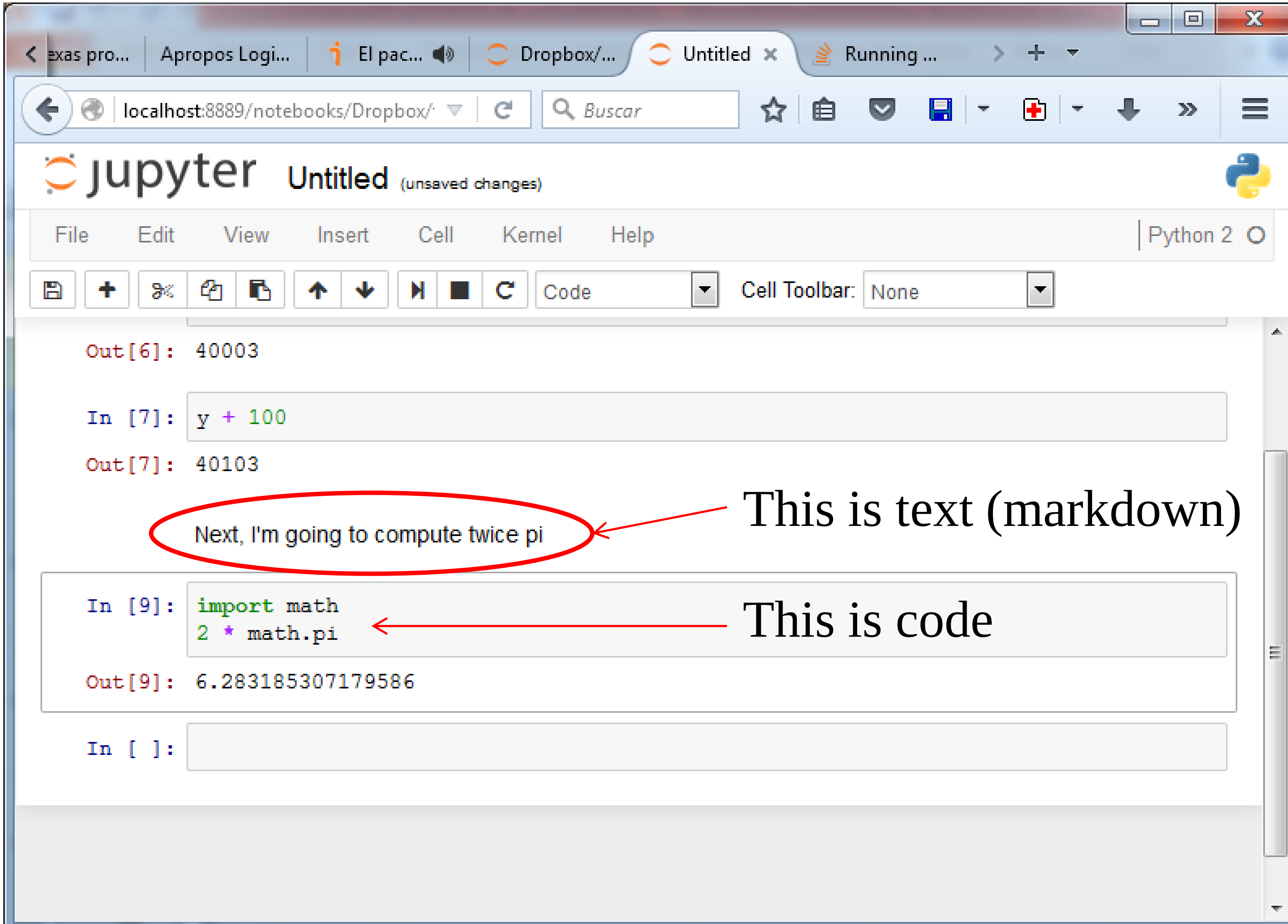
- If you want the changes to propagate to the following cells, you have to execute all of them again.



- In a Python notebook, you can mix text, python commands and results, by changing the cell type



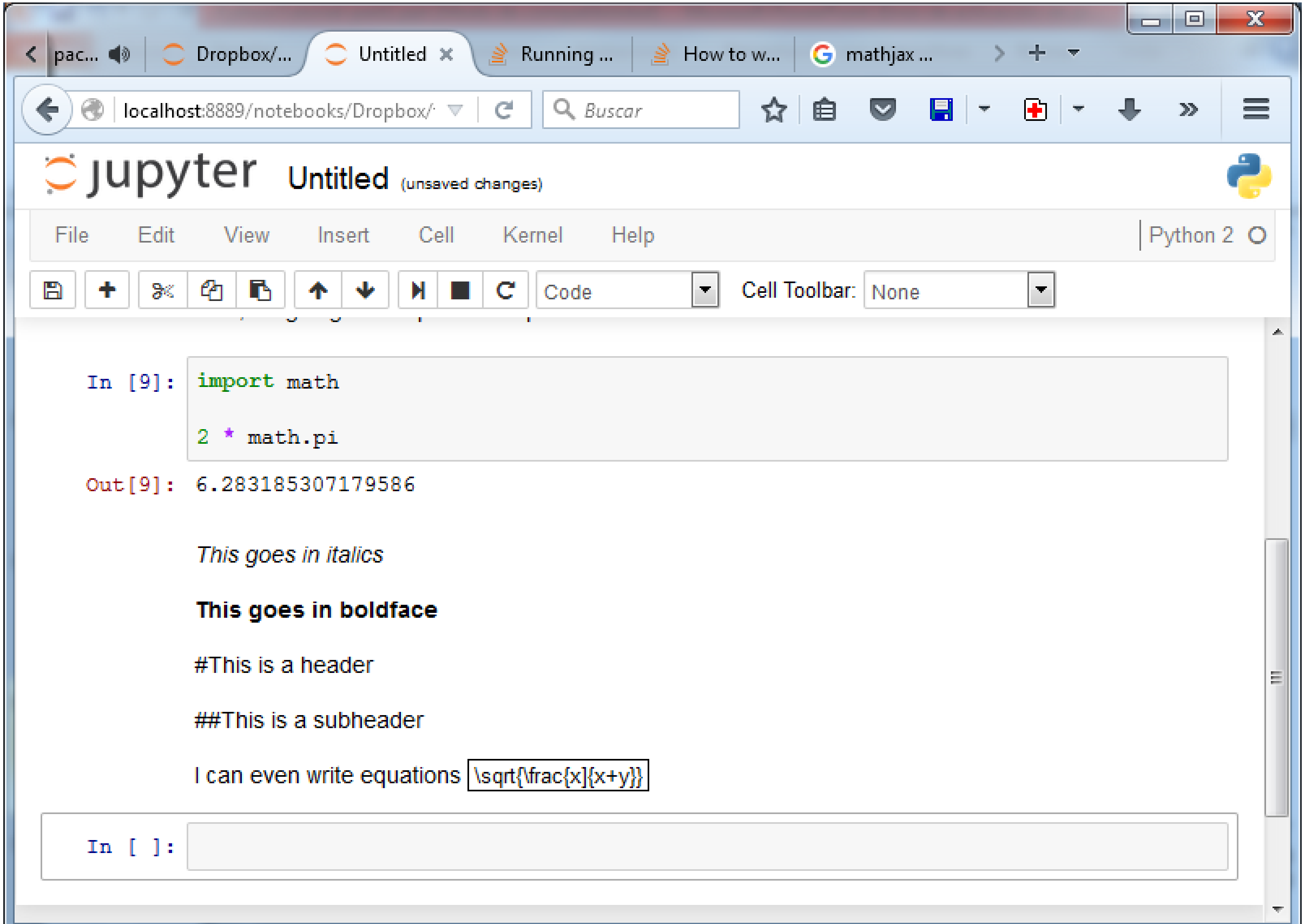
- Text mixed with code



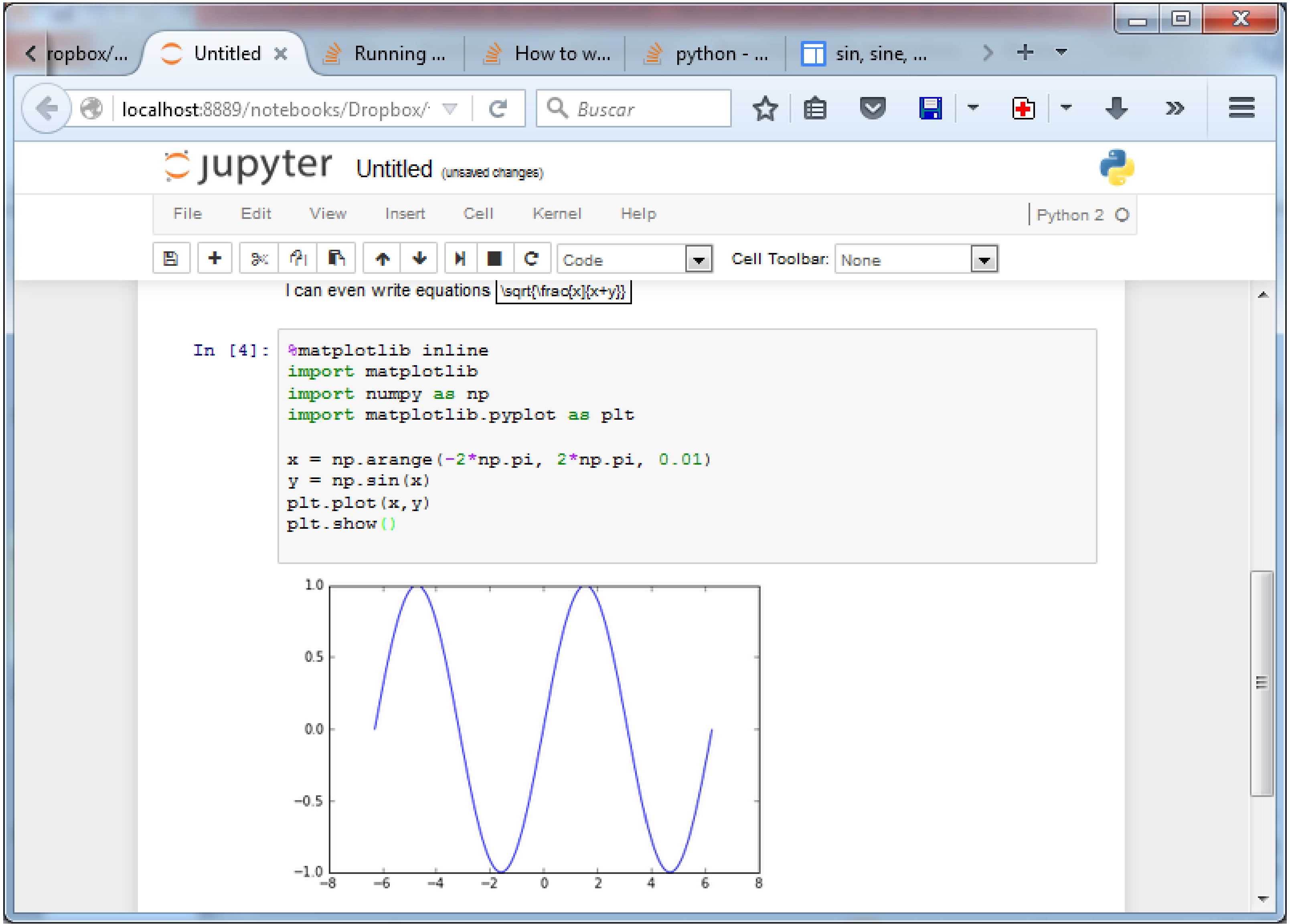
Markdown

- Markdown is a language to format text:
  - *this goes in italics*
  - **this goes in boldface**
  - #This is a header
  - ##This is a subheader
  - I can even write equations (in LaTeX):
    - $\sqrt{\frac{x}{x+y}}$

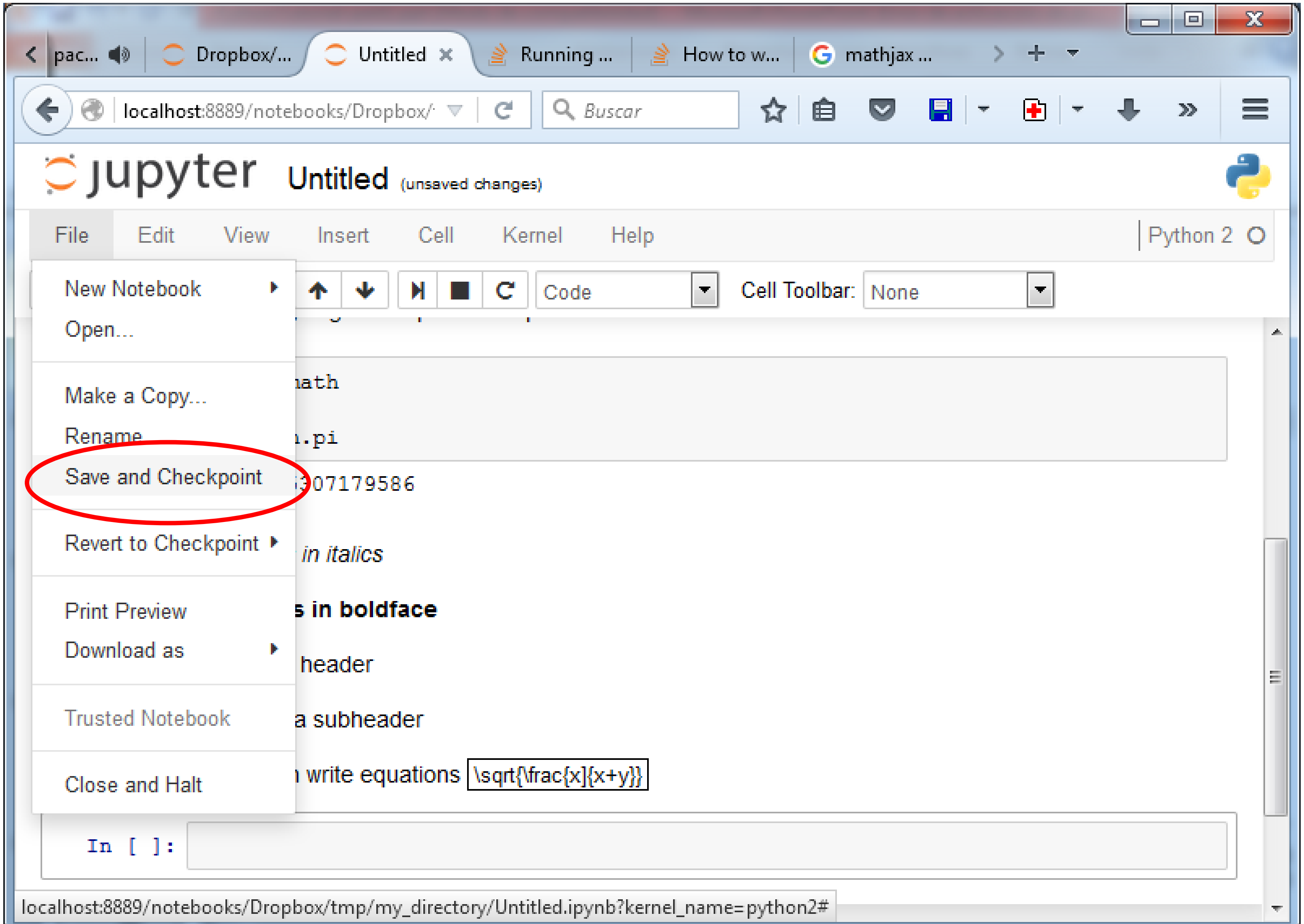
- Markdown


pac...
Dropbox/...
Untitled
Running ...
How to w...
mathjax ...
localhost:8889/notebooks/Dropbox/
Buscar
jupyter
Untitled (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Code
Cell Toolbar:
None
In [9]:
import math
2 * math.pi
Out[9]:
6.283185307179586
This goes in italics
This goes in boldface
#This is a header
##This is a subheader
I can even write equations \sqrt{\frac{x}{x+y}}
In [ ]:

You can even embed plots

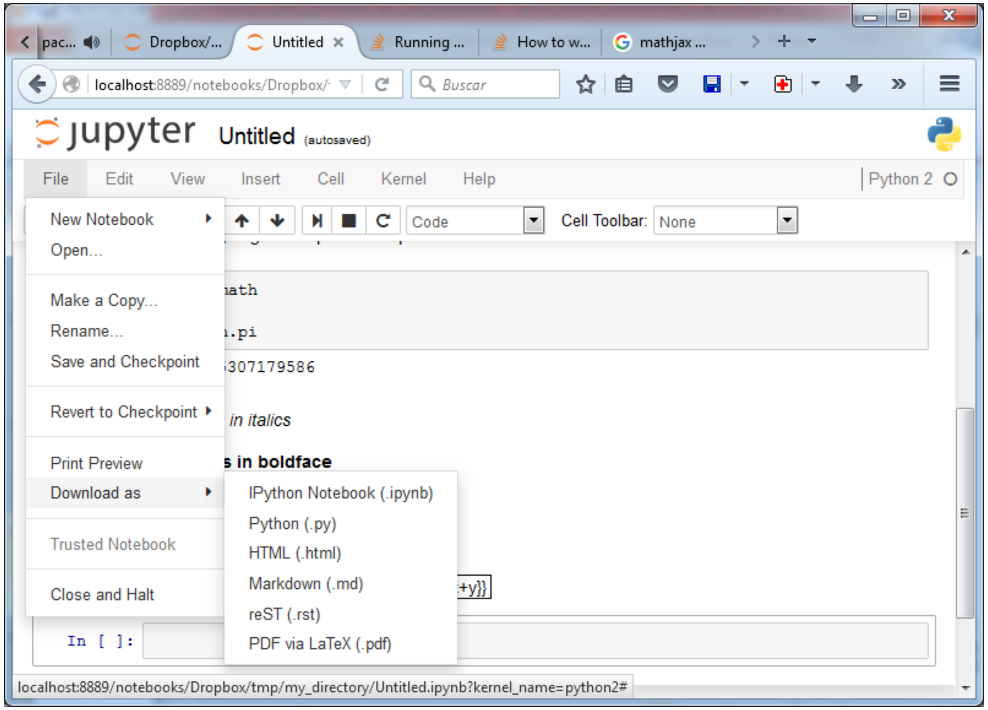


Saving the notebook



Download the notebook

- In several formats: (filename can be changed in File/Rename)
  - Python notebook: it can be loaded again as a notebook
  - Python script: this is a text file containing the sequence of Python commands. Text is also stored as comments (#)
  - html: it can be loaded later in a browser
  - pdf (it might not work because it requires LaTeX)



Etc.

- In order to finish the notebook:
  - File / close and finish
- Jupyter notebooks have more options but you can explore them yourselves

The Python Programming Language: Data Types

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print('print me‘)
print me
>>>
```

Help and comments

help(“print”)

This is a comment

print('Hello world')

Importing Modules

- Python modules are equivalent to R libraries
- Sometimes, some functions are not directly available in Python
- They are included in modules
- Modules have to be imported in order to use its functions
- Example: '+' is included in base Python, but square root (*sqrt*). *sqrt* is included in module math

Importing Modules

If we try to use *sqrt*, we get an error:

```
In [1]: sqrt(2)
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
<ipython-input-1-40e415486bd6> in <module>()
----> 1 sqrt(2)



NameError: name 'sqrt' is not defined
```

Importing Modules

- Let's import module *math*, and use the *sqrt* function within this module, by means of the dot (.) notation
- Modules are similar to R libraries

```
In [2]: import math


In [3]: math.sqrt(2)
Out[3]: 1.4142135623730951
```

Importing Modules

- Sometimes, it is useful to import a function from a library, rather tan the whole library.
- In that case, it is enough to use the name of the function
- Several functions can be imported at the same time

```
In [2]: from math import sqrt, floor


In [3]: sqrt(2)
Out[3]: 1.4142135623730951
```

Importing Modules

- Modules can be given aliases (shorter names for the module)

```
In [2]: import numpy as np
In [3]: np.sqrt(2)
```

The print Statement

- It can be used to print results and variables
- Elements separated by commas print with a space between them

In [**6**]: print('Hello')
Hello

In [**7**]: print('Hello', 'There')
Hello There

Example

- Modules contain functions, but also constants, like pi
- Import module math, assign 2*pi to variable my_pi, and print the result

```
import math

my_pi = math.pi

print(my_pi)
3.141592653589793
```

Variables

- The variable is created the first time you assign it a value
- Everything in Python is an object

```
>>> x = 12
>>> y = " lumberjack "
>>> x
12
>>> y
' lumberjack '
```

- Múltiple assignments (in parallel):

```
In [8]: a = 3
In [9]: b = 4
In [10]: a, b = a+b, a-b
In [11]: a, b
Out[11]: (7, -1)
```

Object types in Python

- Atomic:
  - numbers (in R: numeric)
  - booleans (true, false) (in R: logical)
- Container: (contains other elements)
  - Sequences:
    - Strings: “Hello World!” (in R: a string is atomic, not a sequence)
    - Lists: [1, 2, “three”] (in R: “list”)
    - Tuples: (1, 2, “three”) (not in R)
  - Sets: {'a', 'b', 'c'} (not in R)
  - Dictionaries: {“R”: 51, “Python”: 29} (not in R)

Object types in Python with numpy module

- Container:
  - Important: in Python, unlike R, arrays (and matrices) are not a basic type. It is necessary to use a module
  - Vectors and matrices: (in R: vector, matrix)

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Object types in Python with Pandas module

- Container:
  - Important: in Python, unlike R, dataframes are not a basic type. It is necessary to use a module

  - Dataframes:

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Object types in Python

- Atomic: **numbers**, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Numbers

- integer: 12345, -32
- Long integer: 999999999L
- float: 1.23, 4e5, 3e-4
- octal: 012, 0456
- hex: 0xf34, 0X12FA
- complex: 3+4j, 2J, 5.0+2.5j

Operations with numbers:

- +, -, *, /
- **: power
- // integer division
- % division remainder
- …

```
>>> 123 + 222                # Integer addition
345
>>> 1.5 * 4                  # Floating-point multiplication
6.0
>>> 2 ** 100                 # 2 to the power 100
1267650600228229401496703205376
```

Object types in Python

- Atomic: numbers, **booleans (true, false),** …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Booleans

Whether an expression is true or false

•Values: True, False

<u>Comparisons:</u> ==, <=, >=, !=, …

In [**18**]: 3 == 3
Out[**18**]: True

In [**19**]: 3 == 4
Out[**19**]: False

In [**20**]: 3 < 4
Out[**20**]: True

In [**21**]: "aa" < "bb"
Out[**21**]: True

<u>Combinations:</u> and, or, not

(in R: &&, ||, !)

In [**26**]: (3 == 3) and (3 < 4)
Out[**26**]: True

In [**27**]: (3 == 3) or (3 < 4)
Out[**27**]: True

In [**28**]: not((3 == 3) or (3 < 4))
Out[**28**]: False

Booleans

- Notes:
  - 0 and None are false
  - Everything else is true
  - True and False are just aliases for 1 and 0 respectively

In [**14**]: 1 and 0
Out[**14**]: 0
In [**15**]: 1 or 0
Out[**15**]: 1

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Container:
  - Sequences:
    - **Strings:** “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

String Literals

- They can be defined either with double quotes (“) or single quotes (‘)

```
In [30]: "Hello world"
Out[30]: 'Hello world'

In [31]: 'hello world'
Out[31]: 'hello world'
```

- + is **overloaded** to do concatenation

```
In [16]: x = 'hello'
In [17]: x = x + ' world'
In [18]: print(x)
hello world
```

String Literals: multi-line

- Using triple quotes, strings can be defined across multiple lines

```
>>> s = """ I’m a string
much longer
than the others :)"""

>>> print(s)
 I’m a string
though I am much longer
than the others :)‘
```

Strings: some functions

- **len**(string) – returns the number of characters in the String
- **str**(object) – returns a String representation of the Object

```
In [56]: x = 'ABCDEF'
In [57]: len(x)
Out[57]: 6
In [58]: str(10.1)
Out[58]: '10.1'
```

Strings: some functions

- Some string functions are available only within a module, and the dot (.) notation must be used (similarly to *math.sqrt()*). The module for strings is called *str*.
- For instance, *lower()* and *upper()* are two such functions:

```
In [73]: x = 'It was the best of times, it was the worst of times‘

In [74]: str.lower(x.lower)            # Convert to lowercase
Out[74]: 'it was the best of times, it was the worst of times‘

In [75]: str.upper(x)                  # Convert to uppercase
Out[75]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'
```

String functions

- Other string functions: *count*, *split*, *replace*

```
In [73]: x = 'It was the best of times, it was the worst of times'

In [77]: str.count(x, 'was')               # count counts how many times ‘was’ appears in x
Out[77]: 2

In [79]: print(str.split(x, ' '))           # split splits string x with space ‘ ‘ separator
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']

In [80]: str.replace(x, 'was', 'is') # replace replaces ‘was’ by ‘is’ wherever it appears in x
Out[80]: 'It is the best of times, it is the worst of times'
```

String functions

IMPORTANT!!

- Typically, if you can call a function as module.function(object, other arguments), you can also use another equivalent (but shorter) syntax: object.function(other arguments)
- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments) # We already know this one
- Examples: In [32]: x = 'It was the best of times, it was the worst of times'

```
In [33]: x.lower()
Out[33]: 'it was the best of times, it was the worst of times'

In [34]: # is equivalent to

In [35]: str.lower(x)
Out[35]: 'it was the best of times, it was the worst of times'
```

```
In [36]: x.upper()
Out[36]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'

In [37]: # is equivalent to

In [38]: str.upper(x)
Out[38]: 'IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES'
```

String functions: 2 ways

IMPORTANT!!

- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments)   # We already know this one
- Note: Use dir('   ') to see all methods for strings (dir(3) shows all methods for integers, etc.)
- Examples: In [32]: x = 'It was the best of times, it was the worst of times'

```
In [39]: x.count('was')
Out[39]: 2

In [40]: # is equivalent to

In [41]: str.count(x, 'was')
Out[41]: 2
```

```
In [45]: x.replace('was', 'is')
Out[45]: 'It is the best of times, it is the worst of times'

In [46]: # is equivalent to:

In [47]: str.replace(x, 'was', 'is')
Out[47]: 'It is the best of times, it is the worst of times
```

```
In [42]: print(x.split(' '))
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']

In [43]: # is equivalent to:

In [44]: print(str.split(x, ' '))
['It', 'was', 'the', 'best', 'of', 'times,', 'it', 'was', 'the', 'worst', 'of', 'times']
```

String functions: 2 ways

IMPORTANT!!

- That is, there are two different (but equivalent) ways:
  1. object.function(arguments)
  2. module.function(object, arguments)   # We already know this one

```
In [39]: x.count('was')
Out[39]: 2

In [40]: # is equivalent to

In [41]: str.count(x, 'was')
Out[41]: 2
```

a) Notice that the first way is shorter and you don't need to remember the name of the module (*str*)
b) Only those methods listed with *dir*('was') can be used

Note about replace

- Be careful, *replace()* does not modify the object (but some methods do! modify the object)

```
In [31]: x='It was the best of times, it was the worst of times'
In [32]: x.replace('was', 'is')
Out[32]: 'It is the best of times, it is the worst of times'
In [33]: x
Out[33]: 'It was the best of times, it was the worst of times'
```

Example: string functions

- Split a sentence *x* using both syntax cases:
  - First case: using *split* as a function of *x*: *x.split()*
  - Second case: using *split* as a function of module *str: str.split(x)*

```
In [12]: x = 'It was the best of times, it was the worst of times'
In [13]: x
Out[13]: 'It was the best of times, it was the worst of times'
```

Example: string functions

- Split a sentence *x* using both syntax cases:
  - First case: using *split* as a function of *x*: *x.split()*
  - Second case: using *split* as a function of module *str: str.split(x)*

```
In [12]: x = 'It was the best of times, it was the worst of times'
In [13]: x
Out[13]: 'It was the best of times, it was the worst of times'
```

```
In [14]: # First case
In [15]: x.split(' ')
Out[15]:
['It',
'was',
'the',
'best',
'of',
'times,',
'it',
'was',
'the',
'worst',
'of',
'times']
```

```
In [16]: # Second case: split as function of module str
In [17]: str.split(x, ' ')
Out[17]:
['It',
'was',
'the',
'best',
'of',
'times,',
'it',
'was',
'the',
'worst',
'of',
'times']
```

Positive indices	0	1	2	3	4	5
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’

s = '012345'

Substrings (slicing)

Slicing = obtaining substrings from strings

```
>>> s = '012345'
>>> s[0]
‘0’
>>> s[1]
‘1’
>>> s[3]
'3'
>>> s[1:4]
'123'
```

- Generic slicing sentence: s[start:end:by]
  - Obtain elements from *start* to *(end-1)* with steps of “*by*”
- IMPORTANT*:*
  - *start* begins at 0!!
  - The slice (or substring) includes values from *start* to *end-1*!!!
- *start* >= 0
- *end* < *len(s)*
- *by*: step

To remember: s[:k]+s[k:] = s

Positive indices	**0**	**1**	**2**	**3**	**4**	**5**
Negative indices	-6	-5	-4	-3	-2	-1
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’

Substrings (slicing)

```
>>> s = '012345'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-1]
‘5‘
>>> s[-2]
‘4’
>>> s[-6]
‘0’
```

Generic sentence: s[start:end:by]

Excluding *start* or *end* is the same as index 0 or last index, respectively

s[2:] == s[2:6] == s[2:len(s)]
s[:4] == s[0:4]

Negative indices start at the end of the string

s[-1] == s[5] == s[len(s)-1]
s[-2] == s[4] == s[len(s)-2]
s[-6] = s[-len(s)] == s[0]

Substrings (slicing)

Slicing = obtaining sublists from strings (or from lists)

Positive indices	**0**	**1**	**2**	**3**	**4**	**5**
Negative indices	-6	-5	-4	-3	-2	-1
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’
string2	‘A’	‘B’	‘C’	‘D’	‘E’	‘F’

```
>>> string2 = ‘ABCDEF'
>>> string2[2:]
‘CDEF'
>>> s[:4]
‘ABCDE'
```

```
>>> string2[-1]
‘F‘
>>> string2[-2]
‘E’
>>> string2[-6]
‘A’
```

Substrings (slicing)

Positive indices	0	1	2	3	4	5
Negative indices	-6	-5	-4	-3	-2	-1
s	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’

- Generic sentence: s[start:end:**by**]
- by: step

```
>>> s = '012345'
>>> s[0:4:2]
‘02'
>>> s[0::2]
‘024'
>>> s[-1::-1]
‘543210’
>>> s[-1::-2]
‘531'
```

Get indices from 0 to 3 by 2 (even indices)

Get indices from 0 to end by 2 (even indices)

Get indices from end to beginning by -1 (reverse order)

Get indices from end to beginning by -2 (indices 5, 3, 1 (or equivalently -1, -3, -5)

Exercise

1. Create any string, for instance:

   ‘In a village of La Mancha, the name of which I have no desire to call to mind’

2. Convert it to uppercase:

   'IN A VILLAGE OF LA MANCHA, THE NAME OF WHICH I HAVE NO DESIRE TO CALL TO MIND'

3. Reverse it:

   'DNIM OT LLAC OT ERISED ON EVAH I HCIHW FO EMAN EHT ,AHCNAM AL FO EGALLIV A NI'

4. Obtain another string by keeping one character every four characters (via slicing):

   'D L EENA HOAHAAL L I'

String Formatting: format

- <formatted string>.format(<elements to insert>)

```
>>> "One, {}, three".format(2)
'One, 2, three'
>>> “{}, two, {}".format(1,3)
'1, two, 3'
>>> "{} two {}".format(1, 'three')
'1 two three'
>>> "{0} two {1}".format(1, 'three')
'1 two three‘
>>> "{1} two {0}".format(1, 'three')
'three two 1'
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - **Lists:** [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Lists

- Ordered collection of data
- Elements can be of different types
- Same subset (slicing) operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

Lists are *mutable* (i.e. they can be modified. Strings cannot)

- **x[i] = a** reassigns the ith element to the value a
- Important: variables contain references (pointers) to the object, not the object itself (unlike R)
- Since *x* and *y* point to the same list object, *both* are changed

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
```

Lists: references vs. copies

- If a copy is needed instead of a reference, the copy function can be used (import copy)

Reference: x and y are the same thing

```
In [58]: x = [1, 2, 3]
In [59]: y = x

In [60]: x[1] = 15

In [61]: x
Out[61]: [1, 15, 3]
In [62]: y
Out[62]: [1, 15, 3]
```

Copy: a and b are different things

```
In [63]: import copy
In [64]: a = [1, 2, 3]
In [65]: b = copy.deepcopy(a)

In [66]: a[1] = 15

In [67]: a
Out[67]: [1, 15, 3]
In [68]: b
Out[68]: [1, 2, 3]
```

Lists: Modifying Content

- **x[i:j:k] = b** reassigns the sublist defined by *i:j:k* to list *b*

```
In [7]: x = [0, 1, 2, 3, 4, 5]
In [8]: y = x
In [9]: x[1:3] = ['one', 'two', 'three']
In [10]: x
Out[10]: [0, 'one', 'two', 'three', 3, 4, 5]
In [11]: y
Out[11]: [0, 'one', 'two', 'three', 3, 4, 5]
```

Lists: Modifying Content

- **x.append(12)** inserts element *12* at the end of the list
- **x.extend([13, 14])** extends list [12, 13] at the end of the list
- In both cases **the original list is modified!!!**

```
In [14]: x = [1,2,3]
In [15]: x.append(12)
In [16]: x
Out[16]: [1, 2, 3, 12]
In [18]: x.extend([13, 14])
In [19]: x
Out[19]: [1, 2, 3, 12, 13, 14]
```

- + also concatenates lists, but it does not modify the original list

```
In [20]: y = [1, 2, 3]
In [21]: y  + [13, 14]
Out[21]: [1, 2, 3, 13, 14]
In [22]: y
Out[22]: [1, 2, 3]
```

Reminder: two ways of calling functions on objects

- Let us remember that there are two ways of applying functions to lists (just as with strings):
  1. module.function(object, …)
  2. object.method(…)

```
In [27]: x = [1, 2, 3]
In [28]: list.extend(x, [13, 14])
In [29]: x
Out[29]: [1, 2, 3, 13, 14]

# is equivalent to:

In [30]: x = [1, 2, 3]
In [31]: x.extend([13, 14])
In [32]: x
Out[32]: [1, 2, 3, 13, 14]
```

Lists: deleting elements

- Function *del*:

```
In [33]: x = list(range(10))
In [34]: x
Out[34]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [35]: del(x[1])
In [36]: x
Out[36]: [0, 2, 3, 4, 5, 6, 7, 8, 9]
In [37]: del(x[2:4])
In [38]: x
Out[38]: [0, 2, 5, 6, 7, 8, 9]
```

Sorting lists

- Two ways: *sort()* and *sorted()*
- *list.sort()* changes the list, *sorted()* does not
- reverse=True can be used for reverse order

```
print("ORIGINAL LIST")
print(unique_words)

print("SORTED LIST")
print(sorted(unique_words))
# The original list does not change
print(unique_words)

print("MODIFYING LIST SORT")
print(unique_words.sort())
# The original variable is modified
print(unique_words)
```

```
ORIGINAL LIST
['a', 'best', 'care', 'do', 'ground', 'hobbit', 'hole', 'i', 'in', 'it', 'la', 'lived', 'mancha', 'name', 'not', 'of', 'place',
'remember', 'somewhere', 'the', 'there', 'times', 'to', 'was', 'whose', 'worst']
SORTED LIST
['a', 'best', 'care', 'do', 'ground', 'hobbit', 'hole', 'i', 'in', 'it', 'la', 'lived', 'mancha', 'name', 'not', 'of', 'place',
'remember', 'somewhere', 'the', 'there', 'times', 'to', 'was', 'whose', 'worst']
['a', 'best', 'care', 'do', 'ground', 'hobbit', 'hole', 'i', 'in', 'it', 'la', 'lived', 'mancha', 'name', 'not', 'of', 'place',
'remember', 'somewhere', 'the', 'there', 'times', 'to', 'was', 'whose', 'worst']
MODIFYING LIST SORT
None
['a', 'best', 'care', 'do', 'ground', 'hobbit', 'hole', 'i', 'in', 'it', 'la', 'lived', 'mancha', 'name', 'not', 'of', 'place',
'remember', 'somewhere', 'the', 'there', 'times', 'to', 'was', 'whose', 'worst']
```

Reducing lists

- How to compute, for instance, the sum of the numbers in a list

```
: numbers = [1, 7, 10, 2, 9, 8]
```

```
: # Adding a list of numbers using a loop
  sum = 0
  for number in numbers:
      sum = sum + number

  print(sum)
```

```
37
```

```
: # Adding a list of numbers using reduce
  from functools import reduce
  sum = reduce(lambda x,y:x+y, numbers)

  print(sum)
```

```
37
```

Exercises

- Compute the product of all elements in a list of numbers using reduce
- Concatenate all words in a list of words using reduce

Exercise

- Let us suppose that we have three lists of words. Compute the unique words in the three lists (hint: use reduce and sets)

```
sentences = ["In a hole in the ground there lived a Hobbit".lower().split(' '),
             "It was the best of times it was the worst of times".lower().split(' '),
             "Somewhere in la Mancha in a place whose name I do not care to remember".lower().split(' ')]
```

Solution

```
sentences = ["In a hole in the ground there lived a Hobbit".lower().split(' '),
             "It was the best of times it was the worst of times".lower().split(' '),
             "Somewhere in la Mancha in a place whose name I do not care to remember".lower().split(' ')]
```

```
from functools import reduce

# Transform list of sentences to list of sets
sentences = [set(s) for s in sentences]
# Now, we compute the union of all the sets
unique_words = reduce(lambda x,y: x|y, sentences)
print(unique_words)

# We can convert the set to a list
unique_words = list(unique_words)
# And then, sort the list
unique_words.sort()
# Beware, sort changes the list!
print(unique_words)
```

```
{'somewhere', 'remember', 'hole', 'the', 'mancha', 'in', 'ground', 'best', 'not', 'place', 'la', 'times', 'care', 'do', 'of',
'whose', 'name', 'worst', 'lived', 'to', 'a', 'there', 'hobbit', 'i', 'was', 'it'}
['a', 'best', 'care', 'do', 'ground', 'hobbit', 'hole', 'i', 'in', 'it', 'la', 'lived', 'mancha', 'name', 'not', 'of', 'place',
'remember', 'somewhere', 'the', 'there', 'times', 'to', 'was', 'whose', 'worst']
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - **Tuples:** (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ',' is needed to differentiate from the mathematical expression (2)

```
In [44]: x=(1,2,3)
In [45]: x[1:]
Out[45]: (2, 3)
In [46]: (2,)
Out[46]: (2,)
In [47]: (2)
Out[47]: 2
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - Sets: {'a', 'b', 'c'}
  - **Dictionaries:** {“R”: 51, “Python”: 29}

Dictionaries

- A set of key-value pairs. A key can be any non-mutable object (such as strings, numbers, or **tuples** of non-mutable objects).
- Dictionaries are *mutable*
- Example number of bottles of different drinks
- Access and modification by key

```
In [47]: d = {'milk': 3, 'beer': 21, 'olive oil': 2}
In [48]: d
Out[48]: {'beer': 21, 'milk': 3, 'olive oil': 2}
In [49]: d['milk']
Out[49]: 3
In [50]: d['milk'] = 4
In [51]: d
Out[51]: {'beer': 21, 'milk': 4, 'olive oil': 2}
```

Dictionaries: Add/Delete

- Assigning to a key that does not exist adds an entry:

```
In [52]: d['coffee'] = 3
In [53]: d
Out[53]: {'beer': 21, 'coffee': 3, 'milk': 4, 'olive oil': 2}
```

- Elements can be deleted with *del* (like with lists)

```
In [54]: del(d['beer'])
In [55]: d
Out[55]: {'coffee': 3, 'milk': 4, 'olive oil': 2}
```

Dictionaries

- Obtaining keys and values as lists

```
d = {'milk': 3, 'beer': 21, 'olive oil': 2}
print(d)
```

```
# We can get the list of values
values = list(d.values())
print(values)

# We can get the list of keys
keys = list(d.keys())
print(keys)
```

```
[3, 21, 2]
['milk', 'beer', 'olive oil']
```

Iterating over dictionaries

```
# We can iterate through all elements in a dictionary
for key in d.keys():
    print(key + " " + str(d[key]))
```

```
milk 3
beer 21
olive oil 2
```

```
# We can iterate through all elements in a dictionary
for key,value in d.items():
    print(key + " " + str(value))
```

```
milk 3
beer 21
olive oil 2
```

Default Dictionaries

- It is a dictionary but it is able to return a default value when the key does not exist in the dictionary

```
: d = {'milk': 3, 'beer': 21, 'olive oil': 2}
  print(d['milk'])
  print(d["potatoe"])
```

```
3

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-82-bd07e320ceaa> in <module>()
      1 d = {'milk': 3, 'beer': 21, 'olive oil': 2}
      2 print(d['milk'])
----> 3 print(d["potatoe"])

KeyError: 'potatoe'
```

```
: from collections import defaultdict
  dd = defaultdict(lambda: 0, {'milk': 3, 'beer': 21, 'olive oil': 2})
  print(dd['milk'])
  print(dd["potatoe"])
```

```
3
0
```

Object types in Python

- Atomic: numbers, booleans (true, false), …
- Compound:
  - Sequences:
    - Strings: “Hello World!”
    - Lists: [1, 2, “three”]
    - Tuples: (1, 2, “three”)
  - **Sets**: {'a', 'b', 'c'}
  - Dictionaries: {“R”: 51, “Python”: 29}

Sets

- Sets are like lists, but they only contain unique elements

```
basket = set(['apple', 'orange', 'apple', 'pear', 'orange', 'banana'])
print(basket)
```

```
{'banana', 'pear', 'apple', 'orange'}
```

```
# Checking membership
print('orange' in basket)
print('crab' in basket)
```

```
True
False
```

```
# Sets contain only unique elements
basket = set(['apple', 'apple', 'orange', 'apple', 'pear', 'orange', 'banana'])
print(basket)
```

```
{'banana', 'pear', 'apple', 'orange'}
```

- Any sequence can be used to créate a set, such as strings

```
set1 = set('abracadabra')
print(set1)
```

```
{'b', 'r', 'c', 'd', 'a'}
```

Sets

```
Operations on sets: set difference, union, intersection
```

```python
basket1 = set(['apple', 'orange', 'apple', 'pear', 'orange', 'banana'])
basket2 = set(['apricot', 'coconut', 'apple', 'pear', 'lemon'])
print("Union")
print(basket1 | basket2)

print("Intersection")
print(basket1 & basket2)

print('Set difference')
print(basket1 - basket2)

print('Symmetric set difference = A|B - A^B')
print(basket1 ^ basket2)
print(set((basket1|basket2) - (basket1 & basket2)))
```

```
Union
{'apple', 'pear', 'apricot', 'coconut', 'lemon', 'banana', 'orange'}
Intersection
{'pear', 'apple'}
Set difference
{'banana', 'orange'}
Symmetric set difference = A|B - A^B
{'lemon', 'banana', 'orange', 'apricot', 'coconut'}
{'lemon', 'banana', 'orange', 'apricot', 'coconut'}
```

Sets. Exercise

- Use sets to:
  - Compute the unique letters in strings “abracadabra” and “alacazam”
  - Compute the letters that are in “abracadabra” but not in “alacazam”

Data Type Summary

- Lists, Tuples, and Dictionaries are containers that can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references, but copies can be made

The Python Programming Language: Flow Control

Topics

1. If … then … else
2. Loops:
   - While condition …
   - For …
3. Functions
4. High-level functions (map, filter, reduce)

If Statements

```
if condition :
    sentence1
    sentence2
    …
next sentence
```

```
if condition :
    sentence1
    sentence2
    …
else :
   sentencea
   sentenceb
   …
next sentence
```

```
if condition :
    sentence1
    sentence2
    …
elif condition3 :
   sentencea
   sentenceb
   …
else :
   sentencex
   sentencey
   …
next sentence
```

Indentation

Sentence that follows the “if” (outside of the “if” block)

Example:

```
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30 :
    y = x + 30
else :
    y = x
print ‘y = ‘, y
```

Result is: ?

If Statements

<u>Example:</u>

```
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30 :
    y = x + 30
else :
    y = x
print 'y = ', y
```

Result is: **y = 60**

Note on indentation

- Python uses indentation instead of braces (or curly brackets) to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This forces the programmer to use proper indentation since the indenting is part of the program!
- Indentation made of four spaces is recommended

Example:

Indentation

```
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print 'y = ', y
```

Sentence that follows the "if" (outside of the "if" block)

Exercise

- Use if to determine whether a number is odd or even, and then print “it’s an odd number” or “it’s and even number”

While Loops

While *condition* is true, execute sentences in the *while block* (*sentence1*, *sentence2*, …)

```
while condition :
    sentence1
    sentence2
    …
Next sentence
(outside while block)
```

```
phrase = ['Somewhere', 'in', 'La', 'Mancha']
index = 0
while index < len(phrase) :
    print phrase[index]
    index = index + 1
print '** Words printed, while finished!!'
```

```
Somewhere
in
La
Mancha
** Words printed, while finished!!
```

For Loops

variable takes succesive values in the *sequence*

```
for variable in sequence :
    sentence1
    sentence2
    …
Next sentence (outside for block)
```

```python
phrase = ['Somewhere', 'in', 'La', 'Mancha']
index = 0
for word in phrase :
    print word
print '** Words printed, "for loop" finished!!'
```

```
Somewhere
in
La
Mancha
** Words printed, "for loop" finished!!
```

Exercise

- Create a list of numbers [0, 1, 3, 4, 5, 6]
- Iterate over this list by using a for loop
  - For each element in the list, print “even” if the number is even and “odd” if the number is odd
- Reminder: a number *x* is even if the remainder of the division by 2 is zero. That is: (x % 2 == 0)
- Once you are done, try with another list:

  [1, 7, 3, 2, 0]

Exercise

- Create a list of numbers [0, 1, 3, 4, 5, 6]
- Iterate over this list by using a for loop
- Add all the numbers together

Exercise

- Use a for and a default dictionary to count words in the hobbit_words sentence

```
hobbit_words = "In a hole in the ground there lived a Hobbit".lower().split(' ')
print(hobbit_words)
```

```
['in', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'hobbit']
```

```python
my_dict = defaultdict(lambda: 0)
for word in words:
    my_dict[word] += 1
```

```python
my_dict
```

```
defaultdict(<function __main__.<lambda>()>,
            {'in': 2,
             'a': 2,
             'hole': 1,
             'the': 1,
             'ground': 1,
             'there': 1,
             'lived': 1,
             'hobbit': 1})
```

For and range

- *range*() is an iterator
- It is useful to iterate over a range of values

```
]: for i in range(5):
       print(i)

0
1
2
3
4
```

```
]: for i in range(3,5):
       print(i)

3
4
```

```
]: for i in range(0,10,3):
       print(i)

0
3
6
9
```

```
s = 'in a hole in the ground there lived a hobbit'
words = s.split(' ')
for i in range(len(words)):
    print('Word number {0} is: {1}'.format(i, words[i]))
```

```
Word number 0 is: in
Word number 1 is: a
Word number 2 is: hole
Word number 3 is: in
Word number 4 is: the
Word number 5 is: ground
Word number 6 is: there
Word number 7 is: lived
Word number 8 is: a
Word number 9 is: hobbit
```

Iterators

- Iterators (such as *range*()) allow to iterate over values (i.e. used in a for loop)
- Iterators are not values, but we can use *list*() to get the values of an iterator

```
# Range is an iterator, not a list of values
a = range(10)
print(a)
```

```
range(0, 10)
```

```
# We can get the list of values from an iterator by using list
print(list(a))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Beware! *range()* returned a list in versión 2.7, but returns an iterator in vesion 3.7. There may be cases in this tutorial where *list(range(a,b))* should have been used, but *range(a,b)* is (wrongly!) used.

Loop Control Statements

break	Jumps out of the closest enclosing loop (or while)
continue	Jumps to the top of the closest enclosing loop (or while)
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
while condition :
    sentence1
    sentence2
    …
else:
    sentencea
    sentenceb
Next sentence
(outside while block)
```

```
for variable in sequence :
    sentence1
    sentence2
    …
else:
    sentencea
    sentenceb
Next sentence (outside
for block)
```

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
number = 14
factor = 2
while factor < number :
    if number % factor == 0 :
        print "Number {} is not a prime number".format(number)
        break
    else:
        factor = factor + 1
else:
    print "Number {} is prime".format(number)
```

```
Number 14 is not a prime number
```

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
number = 13
# Note: range(a,b) produces a list of numbers from a to n-1
print range(2, number)
for factor in range(2,number) :
    if number % factor == 0 :
        print "Number {} is not a prime number".format(number)
        break
else: # this block is executed when the loop for exits without break
    print "Number {} is prime".format(number)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Number 13 is prime
```

Function Definition

"return x" returns the value and ends the function exectution

```
def functionName (argument1, argument2, …) :
    sentence1
    sentence2
    …
```

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

```
max(3,5)
```

```
3
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them

```
def double(x=0):
    return(2*x)
```

```
double()
```

```
0
```

```
double(10)
```

```
20
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
In [7]: def myPrint(a,b,c):
            print a,b,c

In [8]: myPrint(c=10, a=2, b=14)

        2 14 10

In [9]: myPrint(3, c=2, b=19)

        3 19 2
```

Exercise

- Write a Python function that computes the factorial of a number. If the input is negative, print "Error". If there is no input, the function should compute the factorial of zero.

Exercise

- Define a function *myDif* that returns:
  - If (a-b)>0 then (a-b)
  - Otherwise b-a
- Both *a* and *b* should have default values of 0
- You need to use *if*
- Try the following function calls and see what happens:
  - myDif(1,2)
  - myDif(2,1)
  - myDif(2)
  - myDif(b=2,a=1)

Functions are first class objects

- Can be assigned to a variable

  x = max

- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Anonymous Functions

- A lambda expression returns a function object
- The body can only be a simple expression, not complex statements

```
>>> f = lambda x,y : x + y
>>> f(2,3)
5
```

List comprehensions

```
]: def double(x):
       """It multiplies x by 2"""
       return(2 * x)

   def even(x):
       return(x % 2 == 0)

   lst = range(10)
   print("Applying double to all elements in {}".format(lst))
   print([double(a) for a in lst])

   print("Filtering / selecting even elements in {}".format(lst))
   print([a for a in lst if even(a)])
```

```
Applying double to all elements in range(0, 10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Filtering / selecting even elements in range(0, 10)
[0, 2, 4, 6, 8]
```

List comprehensions

- They are equivalent to loops, but more elegant

```
def double(x):
    return(2*x)

def even(x):
    return(x % 2 == 0)
```

```
lst = range(10)
```

The following is a list transformation with a list comprehension (each element is doubled)

```
result = [double(a) for a in lst]
print(result)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The previous list comprehension is equivalent to the following loop:

```
result = []
for element in lst:
    result.append(double(element))
print(result)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

List comprehensions

List comprehensions can also be used for filtering (selecting) elements in a list that fulfill some condition. For instance, the following list comprehension filters all even elements in the list.

```
result = [a for a in lst if even(a)]
print(result)
```

```
[0, 2, 4, 6, 8]
```

The previous list comprehension is equivalent to the following loop.

```
result = []
for element in lst:
    if(even(element)):
        result.append(double(element))

print(result)
```

```
[0, 4, 8, 12, 16]
```

Exercises

1. Use a list comprehension to compute the length of the words in a list

```
hobbit_words = "In a hole in the ground there lived a Hobbit".split(' ')
print(hobbit_words)
```

```
['In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'Hobbit']
```

2. Use a list comprehension to convert all the words in hobbit_words to lowercase
3. Use a list comprehension to filter the positive numbers in a list of numbers

Writing and reading files

In [20]:
```
mySentence = "Number three is {}".format(3)
print(mySentence)
# Now, we open file "myFile.txt" for writing
mf = open("myFile.txt", "w")
# Then we write the sentence
mf.write(mySentence)
# Finally, we close the file
mf.close()

# Now, we open the file for reading
mf = open("myfile.txt", "r")
# We read the whole file into variable sentenceFromFile
sentenceFromFile = mf.read()
# We close the file
mf.close()

# And print the sentence, in order to checke whether it is the original sentence
print(sentenceFromFile)
```

```
Number three is 3
Number three is 3
```

Files: Input

inflobj = open('data', 'r')	Open the file 'data' for input.
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Files: Input

Example for reading the whole file into variable *strings*

my_file = open(“data.txt”, “r”)

strings = my_file.read()

my_file.close()

Files: Input

Example for reading line by line into my_line and then printint it:

```
my_file = open(“data.txt”, “r”)
for line in my_file:
    print(line)
my_file.close()
```

Files: Output

outflobj = open('data', 'w')	Open the file 'data' for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Files: Output

```
in_file = open('data.txt', 'r')
out_file = open('output.txt', 'w')
for line in in_file:
    out_file.write('prefix101-'+line+'\n')
my_file.close()
```

EXTRA

Shallow copy vs. deep copy

- Deep copy: it creates two completely different objects
- Shallow copy: it copies only the references to the objects in the list

Shallow copy vs. deep copy

- Reminder: this is not a copy, xs and ys are exactly the same object:

  ```
  xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
  ys = xs
  ```

- Shallow copy:

  ```
  ys = xs.copy()
  ```

Shallow copy vs. deep copy

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = xs.copy()
>>> xs[0] = [10,20,30]
>>> print(xs)
>>> print(ys)
[[10, 20, 30], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Shallow copy vs. deep copy

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = xs.copy()
>>> xs[0][0] = 10
>>> print(xs)
>>> print(ys)
[[10, 2, 3], [4, 5, 6], [7, 8, 9]]
[[10, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Shallow copy vs. deep copy

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = xs.copy()
>>> xs[0][0] = 10
>>> print(xs)
>>> print(ys)
[[10, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```