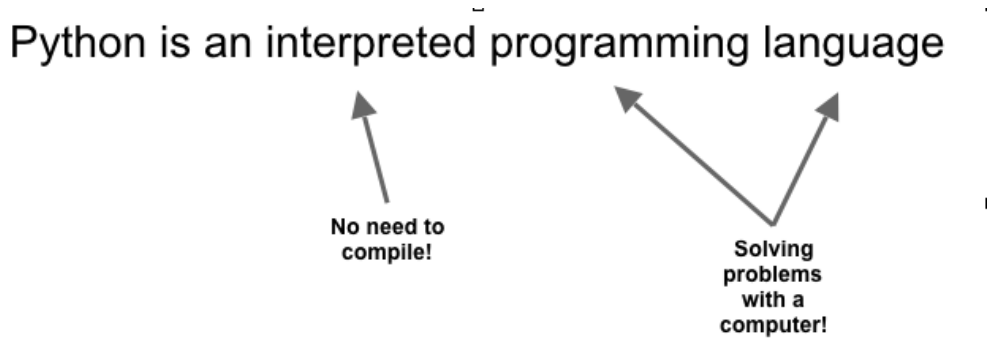
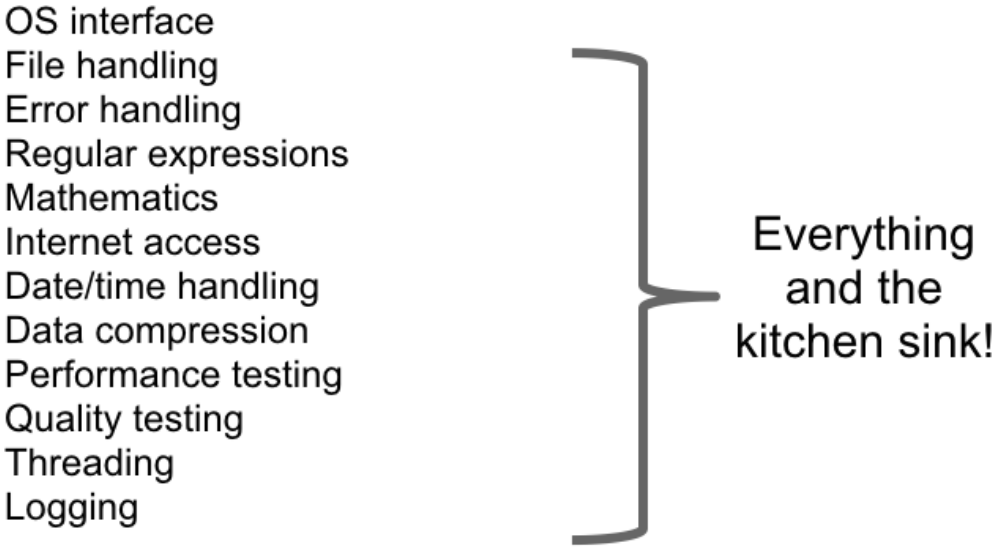
lasp_reu_python_tutorial_day1

June 11, 2016

1 Using Python to investigate data

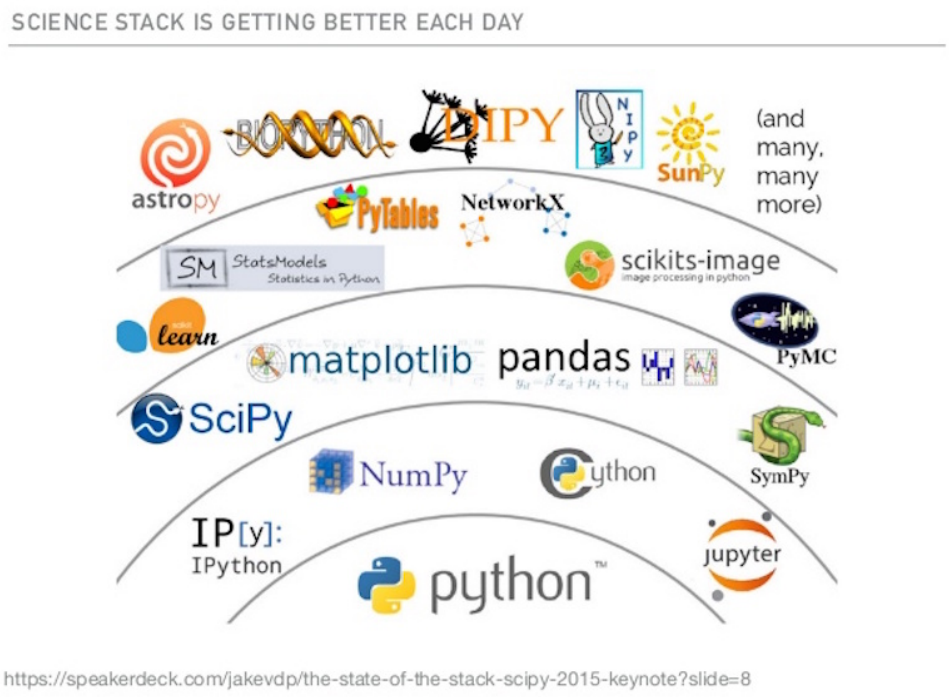


Python overview



Standard library

2 MANY important non-standard packages



Science stack

2.1 Which Python distribution?

- Currently best supported is Anaconda by Continuum.io
- Works on the major three operating systems
- Comes with the most important science packages pre-installed.

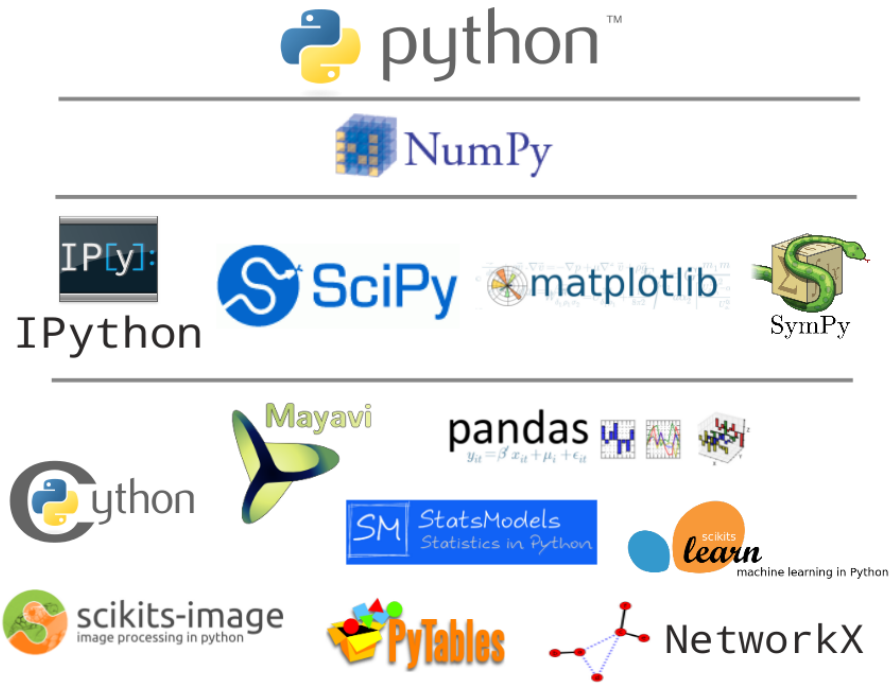
2.2 Getting started

The following launches the Python interpreter in `interactive` mode:

```
$ python
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7 2015, 11:24:55)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world!')
hello world!
```

2.3 To leave Python:

```
>>> <Ctrl-D>
$
```



ecosytem

2.4 Run an existing script file with Python code

This is running Python code `non-interactive`:

```
$ python script_name.py
[...output...]
```

- Python itself can be run interactively, but not many features.
- -> Fernando Pérez, then at CU Boulder, invents IPython, a more powerful interactive environment for Python.

2.5 Launching IPython

Launching works the same way:

```
$ ipython
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7 2015, 11:24:55)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> 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.
Automatic calling is: Smart

In [1]:
```

3 Most important IPython features

- Tab completion for Python modules
- Tab completion for object's attributes ("introspection")
- automatic reload possible of things you are working on.

4 Latest technology jump: IPython notebook (Now Jupyter)

- Cell-based interactivity
- Combining codes with output including plot display **AND** documentation (like this!)
- Very successful. Received twice Sloan foundation funding (here to stay!)
- Became recently language agnostic: JU_lia, PYT_hon, R (and many many more)

5 My recommendation

- Work with IPython for quick things that don't need plots or on slow remote connection
- Work with Jupyter notebook for interactive data analysis and to develop working code blocks
- Put working code blocks together into script files for science production and run "non-interactive"

6 Launching Jupyter notebook

```
$ jupyter notebook
[I 16:27:34.880 NotebookApp] Serving notebooks from local directory: /Users/klay668
[I 16:27:34.880 NotebookApp] 0 active kernels
[I 16:27:34.880 NotebookApp] The Jupyter Notebook is running at: http://localhost:8
[I 16:27:34.881 NotebookApp] Use Control-C to stop this server and shut down all ke
```

- Will launch what is called a "notebook server" and open web browser with dash-board
- This server pipes web browser cells to the underlying Python interpreter.

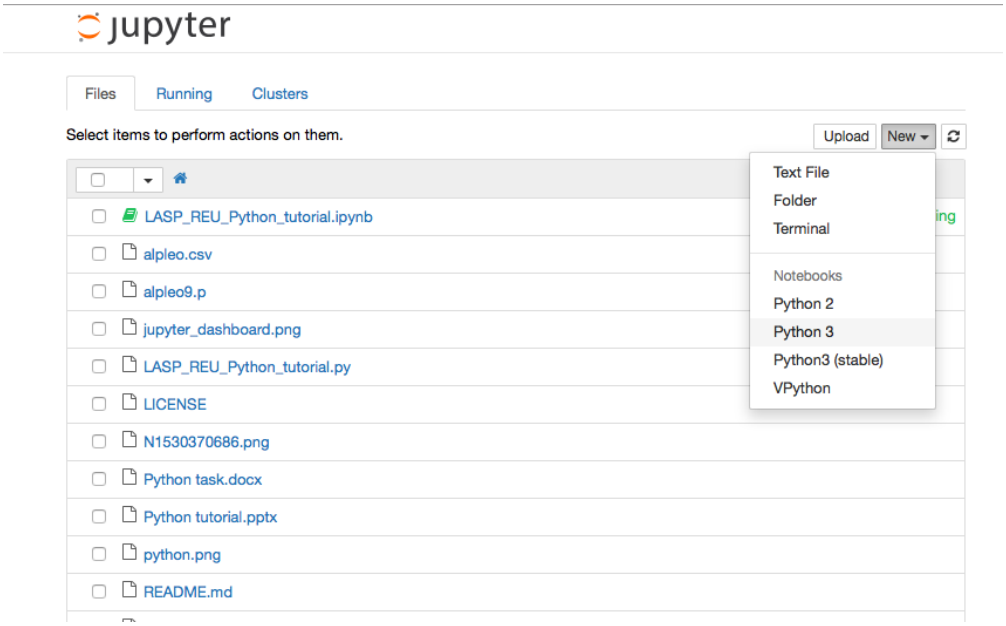
7 And here we are!

7.1 Now let's do some Python!

First, let's look at variables

```
In [1]: a = 5
        s = 'mystring'    # both single and double quotes are okay.
        # new comment
```

Python is `dynamically typed`, meaning I can change the type of any variable at any time:



jupyter dashboard

```
In [3]: a = 'astring'
        print(a)    # not the last line, so if I want to see it, I need to use the p
        s = 23.1
        s  # note that if the last line of any Jupyter cell contains a printable ob

astring


Out[3]: 23.1
```

Python is written in C, therefore uses many of the same technicalities.

For examples, one equal sign = is used for value assignment, while two == are used for equality checking:

```
In [4]: a = 5
        a

Out[4]: 5

In [5]: a == 5

Out[5]: True
```

Storing more than one item is done in `lists`.

A list is only one of Python's `containers`, and it is very flexible, it can store any Python object.

```
In [9]: mylist = [1, 3.4, 'hello', 3, 6]
        mylist
```

```
Out[9]: [1, 3.4, 'hello', 3, 6]
```

Each item can be accessed by an 0-based index (like in C):

```
In [10]: print(mylist[0])
         print(mylist[2])
         len(mylist)

1
hello


Out[10]: 5
```

One can get slices of lists by providing 2 indices, with the right limit being exclusive, not inclusive

```
In [13]: i = 2
         print(mylist[:i])
         print(mylist[i:])

[1, 3.4]
['hello', 3, 6]
```

7.1.1 Sensible multiplication

Most Python objects can be multiplied, in the most logical sense depending on its type:

```
In [14]: a = 5
         s = 'mystring'
         mylist = [1,2]

In [15]: print(5*a)
         print(2*s)
         print(3*mylist)

25
mystringmystring
[1, 2, 1, 2, 1, 2]
```

7.1.2 Conditional branching: if statement

```
In [16]: temp = 80
         if temp > 110:
             print("It's too hot.")
         elif temp > 95 and temp < 110:  # test conditions are combined with `and`
             print("It's okay.")
         else:
             print("Could be warmer.")
         # See how I use double quotes here to avoid ambiguity with single quote in
```

```
Could be warmer.
```

7.1.3 Functions

Functions are called with `()` to contain the arguments for a function.

We already used one: `print()`

Learn about function's abilitys using IPython's help system. It is accessed by adding a question mark to any function name. In Jupyter notebooks, a sub-window will open. When done reading, close it by pressing `q` or clicking the x icon:

```
In [17]: print?
```

Making your own function This is very easy, with using the keyword `def` for "define":

```
In [18]: def myfunc(something):  # note how I don't care about the type here!
             """Print the length of `something` and print itself."""
             print("Length:", len(something))   # all `something` needs to support
             print("You gave:", something)

In [19]: myfunc('mystring')

Length: 8
You gave: mystring


In [20]: myfunc(['a', 1, 2])

Length: 3
You gave: ['a', 1, 2]


In [21]: myfunc(5)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-21-4959b330b503> in <module>()
    ----> 1 myfunc(5)


        <ipython-input-18-aecc323f621a> in myfunc(something)
          1 def myfunc(something):  # note how I don't care about the type here!
          2     """Print the length of `something` and print itself."""
    ----> 3     print("Length:", len(something))   # all `something` needs to suppo
          4     print("You gave:", something)
```

```
TypeError: object of type 'int' has no len()
```

The principle of not defining a required type for a function, but require an `ability` is very important in Python and is called `duck typing`:

> "In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with."

7.2 Loops

Loops are the kings of programming, because they are the main reason why we do programming:

> Execute tasks repeatedly, stop when conditions are fulfilled.

These conditions could be simply that data are exhausted, or a mathematical condition.
2 main loops exist: The more basic `while` loop, and the more advanced `for` loop.

7.2.1 while loops

`while` loops run until their conditional changes from True to False.

The loop is only entered when the condition is True.

Note how in Python sub blocks of code are defined simply by indentation and the previous line ending with a colon `:`.

```
In [26]: i = 0
         while i < 3:  # this is the condition that is being checked.
             print(i, end=' ')
             i = i + 1   # very common statement, read it from right to left!
         print('Done')
```

```
0 1 2 Done
```

```
In [24]: i < 3
```

```
Out[24]: False
```

```
In [25]: i
```

```
Out[25]: 3
```

`while` loops are the most low level loops, they always can be made to work, as you design the interruption criteria yourself.

7.2.2 For loops

`for` loops are designed to loop over containers, like lists.

They know how to get each element and know when to stop:

```
In [27]: mylist = [5, 'hello', 23.1]
         for item in mylist:
             print(item)

5
hello
23.1
```

The "`in`" keyword is a powerful and nicely readable concept in Python.

In most cases, one can check ownership of an element in a container with it:

```
In [28]: 5 in mylist

Out[28]: True
```

7.2.3 The `range()` function

`range()` is very useful for creating lists for you that you can loop over and work with.

It has two different call signatures:

- range(n) will create a list from 0 to n-1, with an increment of +1.
- range(n1, n2, [step]) creates a list from n1 to n2-1, with again a default increment of 1

Negative increments are also okay:

```
In [29]: for i in range(10):
             print(i, end=' ')

0 1 2 3 4 5 6 7 8 9

In [30]: for i in range(2, 5):
             print(i, end=' ')

2 3 4

In [33]: for i in range(0, -5, -1):
             print(i, end=' ')

0 -1 -2 -3 -4

In [34]: range?
```

IMPORTANT

Note, that for memory efficiency, `range()` is not automatically creating the full list, but returns an object called a `generator`.

This is basically an abstract object that knows **HOW TO** create the requested list, but didn't do it yet.

```
In [35]: print(range(10))
```

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

It takes either a loop (as above) or a conversion to a list to see the actual content:

```
In [36]: print(list(range(10)))
```

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

7.2.4 Combine `if` and `for`

Let's combine `for` and `if` to write a mini program. The task is to scan a container of 1's and 0's and count how many 1's are there.

```
In [37]: mylist = [0,1,1,0,0,1,0,1,0,0,1]
         mylist
```

```
Out[37]: [0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1]
```

```
In [38]: one_counter = 0
         for value in mylist:        # note the colon!
             if value == 1:          # note the indent for each `block` !
                 one_counter += 1    # this is the short version of a = a + 1
         print("Found", one_counter, "ones.")
```

```
Found 5 ones.
```

7.2.5 Writing and reading files

Need to get data in and out. In principle, I recommend to use high level readers from science packages.

But you will need to know the principles of file opening nonetheless.

```
In [39]: afile = open('testfile', 'w')
```

```
In [40]: afile.name?
```

```
In [41]: afile.write('some text \n')    # \n is the symbol to create a new line
         afile.write('write some more \n')
```

```
Out[41]: 17
```

The `write` function of the `afile` object returns the length of what just was written.

When done, the file needs to be closed!

Otherwise, the content could not end up in the file, because it was `cached`.

```
In [42]: afile.close()
```

I can call operating system commands with a leading exclamation mark:

```
In [43]: !cat testfile

some text
write some more
```

reading the file One can also use a so called `context manager` and indented code block to indicate to Python when to automatically close the file:

```
In [47]: with open('testfile', 'r') as afile:
             print(afile.readlines())

['some text \n', 'write some more \n']
```

```
In [48]: with open('testfile', 'r') as afile:
             print(afile.read())

some text
write some more
```

7.2.6 Tutorial practice

Now you will practice a bit of Python yourself.

I recommend to join forces in groups of two.

This working technique is called "pair programming": One person is the "driver", typing in code, while the other person is the "navigator", reviewing everything that is being typed in. The roles should be frequently changed (or not, depending on preferences).

This way you can discuss anything that's being worked on.

Next session we will learn how to import all these powerful analysis packages into a Python session, learn about the most import science packages and play with some real data.