



# IPython & Jupyter Notebook

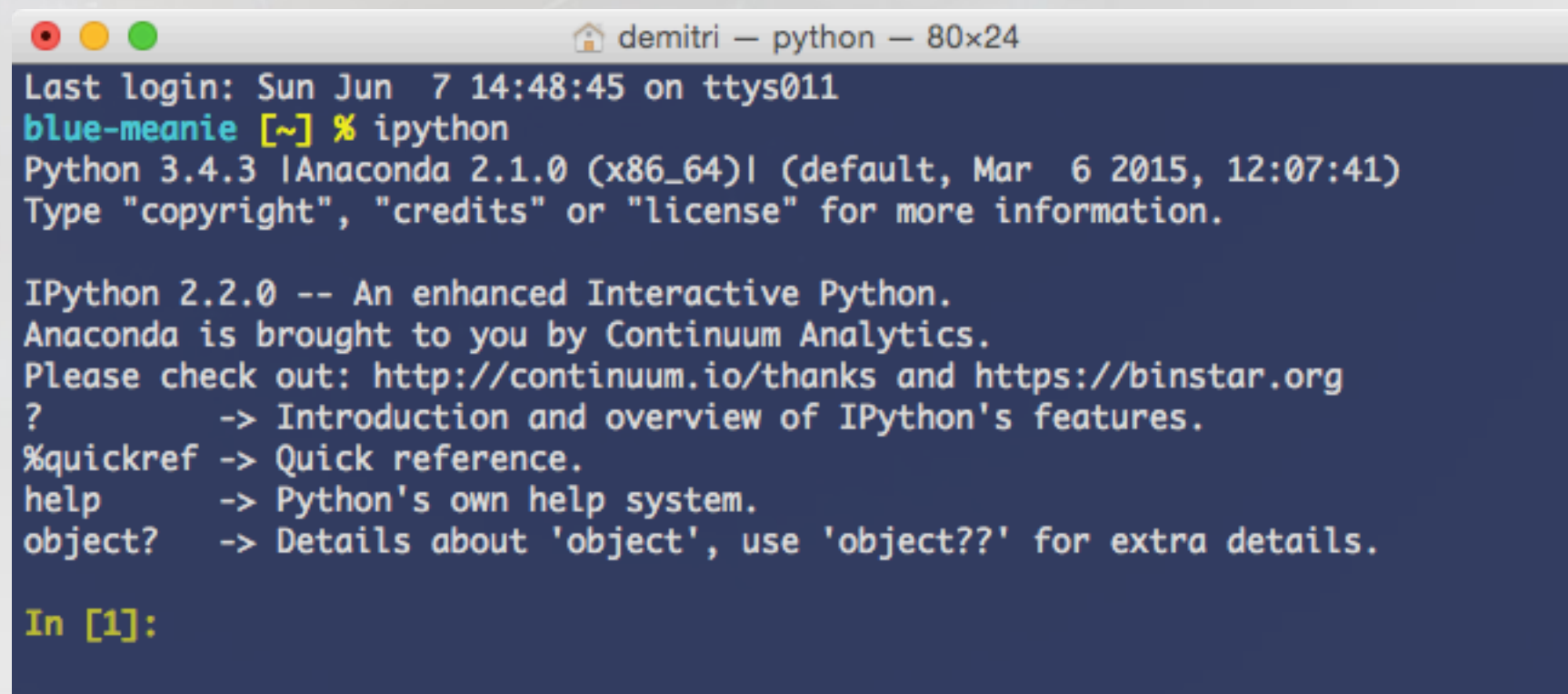
Demitri Muna

4 August 2016

# IPython

IPython is a more feature-rich command line interactive Python environment. It's a good substitute for the regular Python command line interpreter when using it as an interactive environment.

- Access built-in documentation for any class
- Provides command completion based on the methods and properties of the class
- Better debugging facilities

A screenshot of a terminal window titled "demitri — python — 80x24". The terminal shows the command "ipython" being executed, which displays the IPython version (2.2.0) and the Anaconda version (2.1.0). It also shows the last login time and a list of help commands: "?", "%quickref", "help", and "object?". The prompt "In [1]:" is visible at the bottom.

```
demitri — python — 80x24
Last login: Sun Jun  7 14:48:45 on ttys011
blue-meanie [~] % ipython
Python 3.4.3 |Anaconda 2.1.0 (x86_64)| (default, Mar  6 2015, 12:07:41)
Type "copyright", "credits" or "license" for more information.

IPython 2.2.0 -- 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]:
```

# Using IPython

NOTE! IPython and Jupyter Notebook (formerly IPython Notebook) is intended to be used interactively.

One does not typically write software or run software from there environments, rather, they are more for interactive exploration of code and data.

Further documentation:

<http://ipython.org/ipython-doc/stable/interactive/tutorial.html>  
<http://www.pythonforbeginners.com/basics/ipython-a-short-introduction>

# IPython Tab Completion

Create an object in IPython, e.g. a list. Then start to type the command to append a value:

```
In [1]: a = list()
```

```
In [2]: a.app
```

hit tab

```
In [1]: a = list()
```

```
In [2]: a.append
```

IPython is aware of the type of object, which allows command completion. If you don't specify the start of a command, you'll get a list of all available commands:

```
In [1]: a = list()
```

```
In [2]: a.
```

hit tab

```
In [1]: a = list()
```

```
In [2]: a.
```

a.append	a.copy	a.extend
a.insert	a.remove	a.sort
a.clear	a.count	a.index
a.pop	a.reverse	




# IPython Tab Completion

Autocomplete also works with modules...

```
In [4]: from sys import
```

hit tab



...and files, and directory names.

(Basically try hitting tab a lot.)

```
In [4]: from sys import
__egginsert      executable      modules
__plen           exit            path
_clear_type_cache flags          path_hooks
_current_frames  float_info
path_importer_cache
_debugmallocstats float_repr_style  platform
_getframe        getallocatedblocks prefix
_home            getcheckinterval  ps1
_mercurial       getdefaultencoding ps2
_xoptions        getdlopenflags  ps3
abiflags         getfilesystemencoding setcheckinterval
api_version      getprofile         setdlopenflags
argv             getrecursionlimit  setprofile
base_exec_prefix getrefcount         setrecursionlimit
base_prefix      getsizeof           setswitchinterval
builtin_module_names getswitchinterval  settrace
byteorder        gettrace           stderr
call_tracing     hash_info           stdin
callstats        hexversion          stdout
copyright        implementation  thread_info
displayhook      int_info            version
dont_write_bytecode intern          version_info
exc_info          maxsize             warnoptions
excepthook       maxunicode
exec_prefix      meta_path
```

# Explore Documentation

To get a summary of information about an object, type a “?” after it and hit return. This is a quick and powerful way to explore a Python package to discover what’s available. This works for individual methods as well. This information comes from the source code (which we’ll go into later).

```
In [1]: a = list()

In [2]: a?
Type:      list
String form: []
Length:    0
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items

In [3]: a.append?
Type:      builtin_function_or_method
String form: <built-in method append of list object at 0x1037c1e08>
Docstring: L.append(object) -> None -- append object to end

In [4]:
```

Documentation for IPython itself can be accessed by entering “?” alone and hitting return.

# Explore Documentation

The *help* command summarizes the methods of an object's *type*.

```
In [26]: a = [1,2,3,4]
In [27]: help(a)

Help on list object:

class list(object)
| list() -> new empty list
| list(iterable) -> new list initialized from iterable's items
|
| Methods defined here:
|
|   __add__(self, value, /)
|       Return self+value.
|
|   __contains__(self, key, /)
|       Return key in self.
|
|   __delitem__(self, key, /)
|       Delete self[key].
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|
```

# Magic Commands

There are commands that apply specifically to the IPython environment that are *not* Python commands. These are called *magic commands*, and are preceded by a ‘%’ character.

There is a setting called *automagic* that will let you type the magic commands without the leading ‘%’ character. Note that this command is turned **on** by default.

The *lsmagic* command lists all magic commands.

```
In [4]: lsmagic
Out[4]:
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.
```



# Magic Commands

Of course, you can just ask IPython what each magic command does.

```
In [5]: ?lsmagic
Type:      Magic function
String form: <bound method BasicMagics.lsmagic of <IPython.core.magics.basic.BasicMagics
object at 0x10379eeb8>>
Namespace: IPython internal
File:      /usr/local/anaconda/lib/python3.4/site-packages/IPython/core/magics/basic.py
Definition: lsmagic(self, parameter_s='')
Docstring:  List currently available magic functions.
```

Use the %quickref command to display a “quick reference card” of these commands.

```
In [6]: %quickref
```

# Useful Magic Commands

<code>%who</code>	list all defined variables and packages that have been imported
<code>%edit filename</code>	open the specified filename in \$EDITOR (as defined in your shell)
<code>%hist</code>	lists your command history
<code>%hist -g string</code>	search history for commands containing the specified string
<code>%reset</code>	resets the environment (removes all defined variables, imported packages, etc.)
<code>%run filename</code>	executes the specified file into the current environment
<code>%rerun n</code>	Rerun the <i>n</i> th command in the history. Pass “-n” to the history command to display the numbers.

Many shell commands work as magic commands: `pwd`, `cd`, `cp`, `mv`, `more`, `man`, `ll`, `less`, etc.

# Executing Shell Commands

Shell commands can be executed directly in IPython by preceding them with “!”. The output can also be captured into a Python variable.

```
In [16]: !ls
Accounts      GameKit      PubSub
Address Book  Plug-Ins     Google       QuickLook
Application Scripts Graphics      Safari
Application Support Group Containers Saved Application State
Assistants    Icons        Saved Searches
Audio         IdentityServices Screen Savers

In [16]: files = !ls

In [16]: files
Out[16]:
['Applications',
 'Library',
 'Network',
 'System',
 'User Information',
 'Users',
 'Volumes',
 'bin',
 'cores',
 ...]
```

# Debugging

IPython is useful when debugging. Normally when a Python script is run from the command line, it exits altogether if it crashes.

Run your script as below to drop into an IPython prompt where you can continue working/ debugging at the point where the code failed.

```
% ipython --pdb my_script.py
```



# Jupyter Notebook

- Interactive, visual front end to Python
- Mathematica-like interface
- Supports text, code, plots, LaTeX, graphics, movies... most rich media
- Text is rendered in the Markdown markup language
- The notebook is cell-based
- The Notebook runs in the browser from a lightweight web server
- Notebooks are files that can be shared via URLs, emailed, hosted on other servers
- Particularly good for presenting data while still allowing exploration and interaction

# Starting Jupyter Notebook

Jupyter Notebook is started from the command line with this. Typically run this from the directory where you want to keep your files (but this is not strictly necessary).

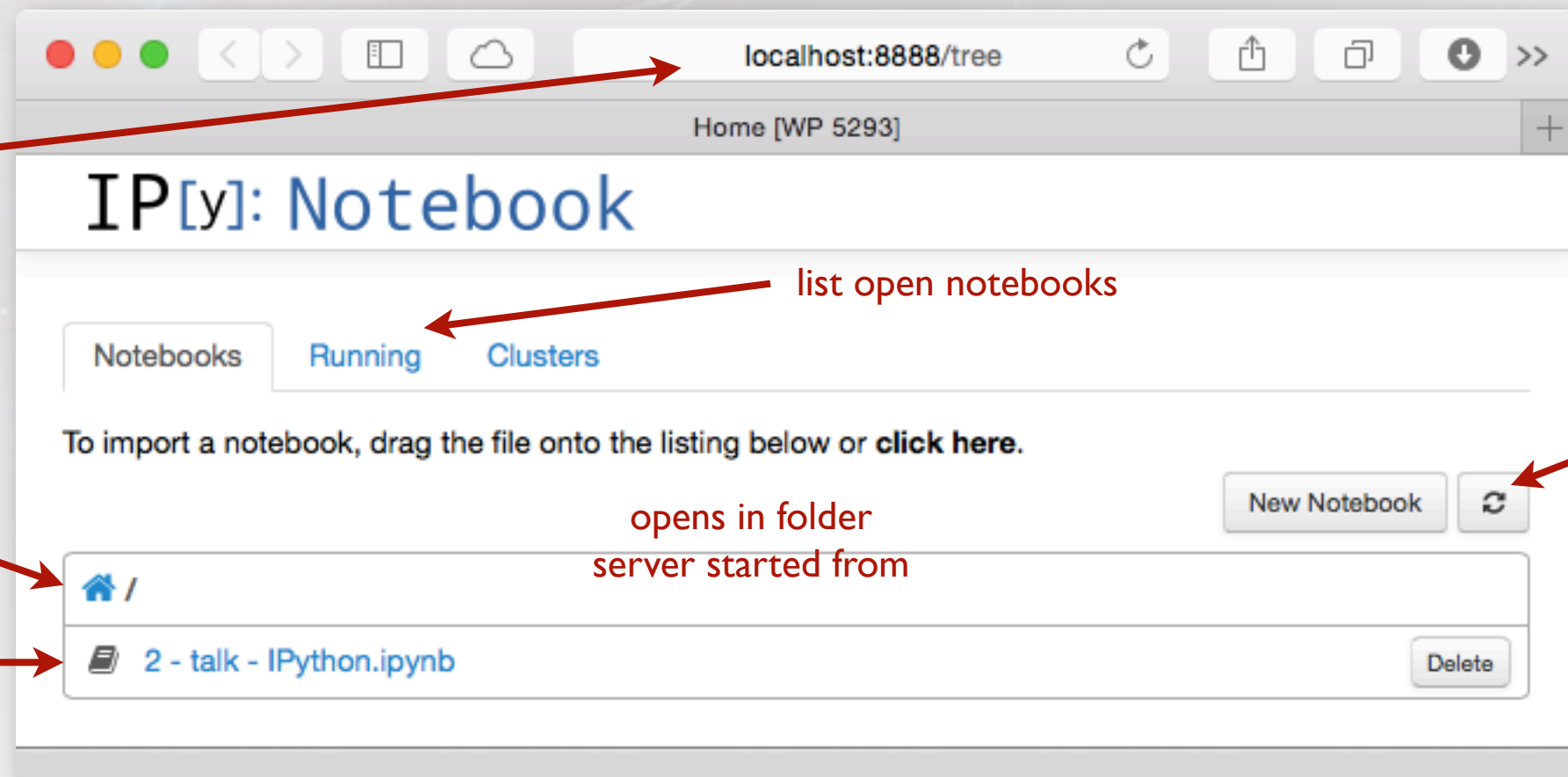
```
% jupyter notebook
```

This starts up a web server in the background and will open in your web browser:

if you close your browser, this is the address to connect back to the server:  
<http://localhost:8888>

can only navigate below the folder you started in

click to open



list open notebooks

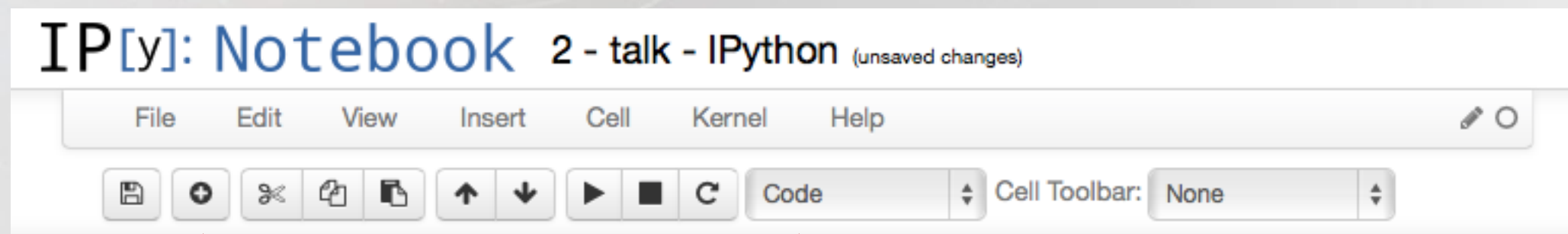
opens in folder  
server started from

refresh page (e.g.  
when adding a new  
notebook)

# IPython Notebook

The IPython Notebook interface is a series of cells. Cells can be

- Python code
- Markdown-formatted text
- Raw (output cell)
- Headings



create a new cell

change the type of cell

# Creating & Editing Cells

When a cell is first selected, it is in an editable mode:

```
This is a heading cell.
```

When finished, type Shift+Enter to render the cell.

**This is a heading1 cell.**

Double click the cell to edit again. This is true for the heading, code, and Markdown cell types (you can't edit an output).

Equations are supported as LaTeX in Markdown-type cells:

```
Equations!  $\sqrt{x^2 + \beta^2} = \alpha$ 
```

shift+enter

Equations!  $\sqrt{x^2 + \beta^2} = \alpha$

not perfect though...



# Code Cells

Text entered in to code cells will be interpreted as Python code.

```
In [ ]: range(10) + 2
```

currently selected  
cell has a border

To execute the code:

Shift+Enter : execute code, move to the next cell

Control+Enter: execute code, keep cursor in the same cell

```
In [8]: print([x for x in range(10)])  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

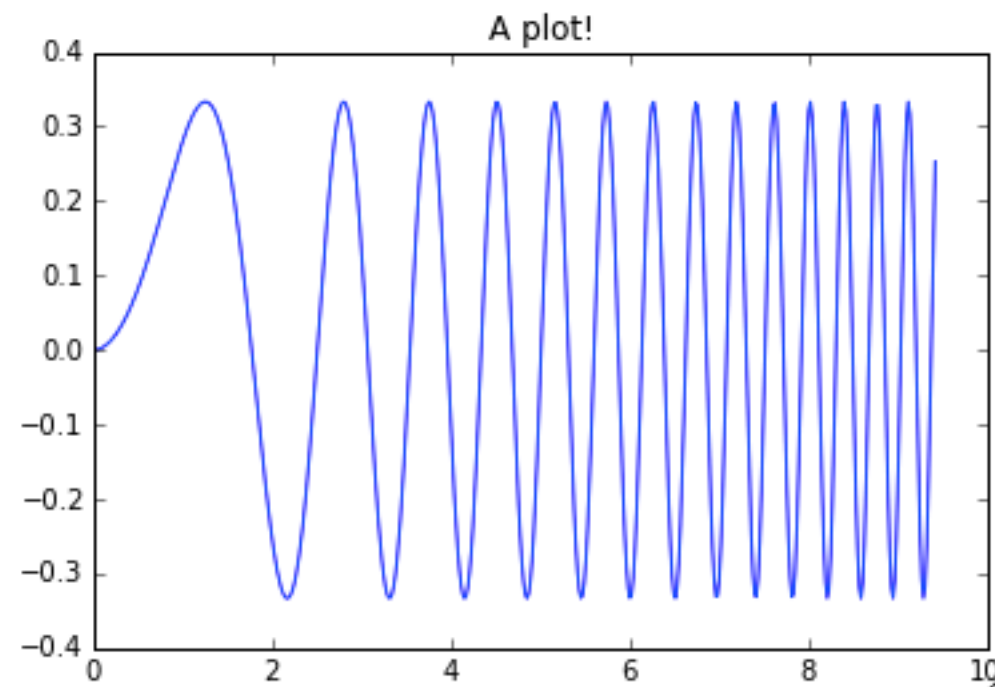
# Inline Plots

Matplotlib plots can be rendered inline, but first require a magic command in the notebook:

```
In [7]: %matplotlib inline
```

```
In [12]: from matplotlib.pyplot import *  
import numpy as np  
from math import pi  
x = np.linspace(0, 3*pi, 500)  
plot(x, np.sin(x**2)/3)  
title("A plot!")
```

```
Out[12]: <matplotlib.text.Text at 0x1132d77f0>
```



imports need only be made once per notebook (not per cell), before they are used

# Package/Function Completion

Once you import a package, IPython has can read all of its functions/methods (i.e. everything in its namespace). You can use the tab key to show the options available:

```
In [ ]: scipy.
```

tab ↓

```
In [ ]: scipy.ALLOW_THREADS  
scipy.BUFSIZE  
In [ ]: scipy.CLIP  
scipy.ComplexWarning  
scipy.DataSource  
In [ ]: scipy.ERR_CALL  
scipy.ERR_DEFAULT  
scipy.ERR_IGNORE  
In [1]: scipy.ERR_LOG  
scipy.ERR_PRINT
```

(esc to close)

```
In [ ]: scipy.
```

# Package/Function Completion

Similarly:

```
In [ ]: scipy.int
```

tab ↓

```
In [ ]: scipy.int0  
        scipy.int16  
        scipy.int32  
In [ ]: scipy.int64  
        scipy.int8  
In [ ]: scipy.int_  
        scipy.int_asbuffer  
        scipy.intc  
In [1]: scipy.integer  
        scipy.interp  
In [ ]: scipy.int|
```

Once the completion window is open, you can continue to type and the list will live update.



# Inline Documentation

Documentation for a particular function is available while you are typing it. For example, start writing a function:

```
In [ ]: range(10,
```

shift+tab



```
In [ ]: range(10,
```

```
Docstring:  
range(stop) -> range object  
range(start, stop[, step]) -> range object
```

^ + x

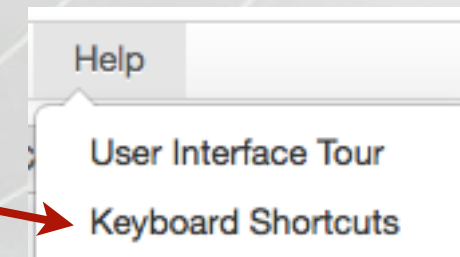
(esc to close)

This is only available after the open parenthesis has been typed.

# Keyboard Shortcuts

There are several keyboard shortcuts available. Display the full list of them with `cntrl-M h`

Or if you hate emacs as much as I do, from the menu:



Note that there are two modes for keyboard commands:

The IPython Notebook has two different keyboard input modes. **Edit mode** allows you to type code/text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level actions and is indicated by a grey cell border.

# Reference

<http://ipython.org/ipython-doc/3/notebook/notebook.html>