Wait, IPython can do that?!

Sebastian Witowski

$ whoami

 Python consultant and trainer

 @SebaWitowski

 https://switowski.com/blog

Technical remarks

Here are the slides of my talk:

bit.ly/advanced-ipython

with:

IPython version 7.4

Python version 3.7.2

Motivation

- I’ve been using IPython since version 0.x (over 6 years) ...
- ... and I thought that **everyone** is using it (which is not the case)
- There are many features!
- And today we will talk about the most interesting ones

History of IPython

- IPython is the father of the **Jupyter Project**
- Started in 2001 as **259** lines of code executed at Python's startup, written by Fernando Perez (history of IPython blog post):
  - Numbered prompt
  - Store the output of each command in global variables
  - Load some additional libraries (numerical operations and plotting)
- Interactive prompt → Notebooks → Project Jupyter

This talk is NOT about Jupyter

IPython and Jupyter in Depth: High productivity, interactive Python

https://www.youtube.com/watch?v=VQBZ2MqWBZI

This talk is about IPython

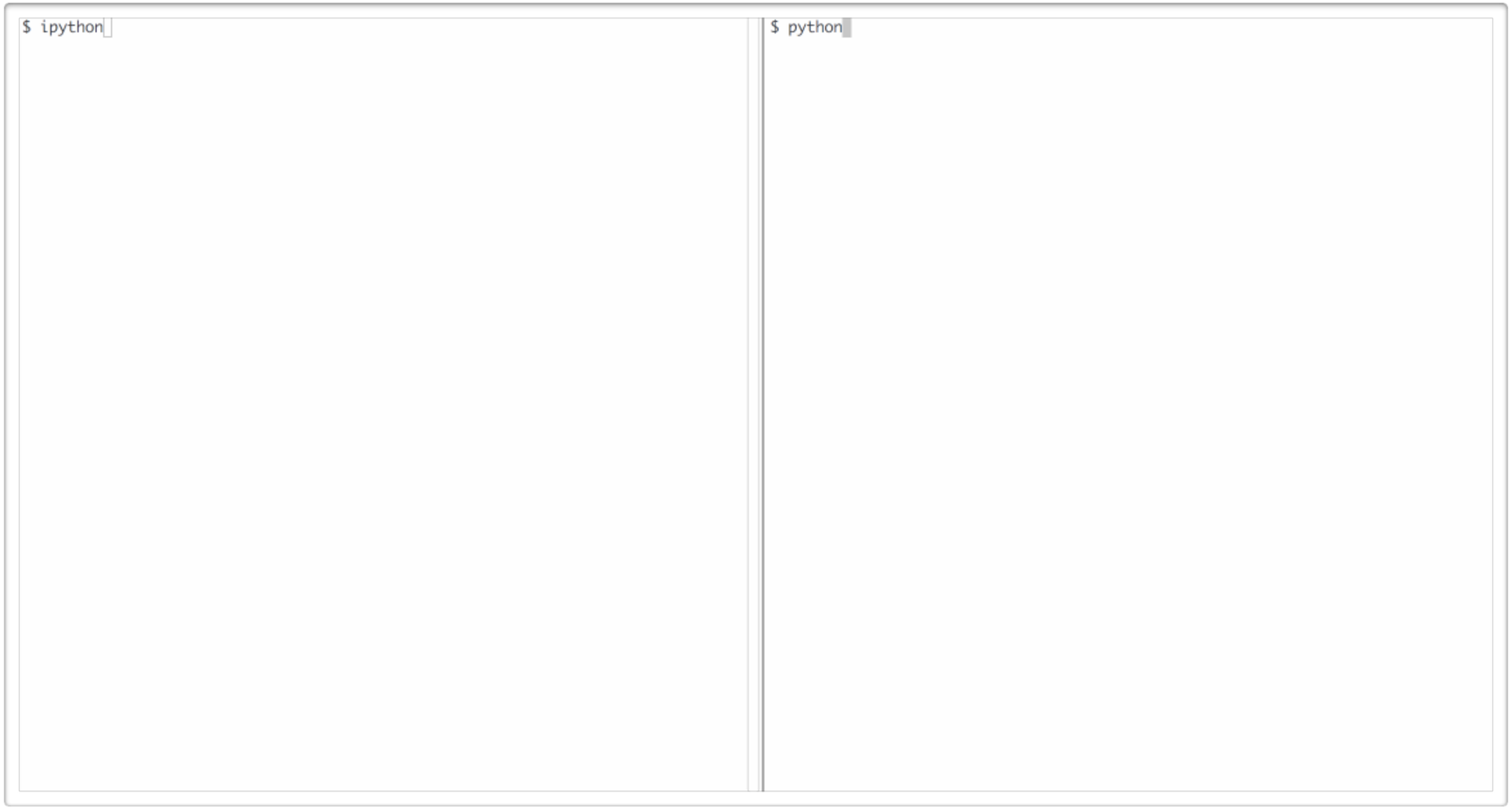
But many of the things will apply to Jupyter as well

IPython REPL

What's a **REPL**?

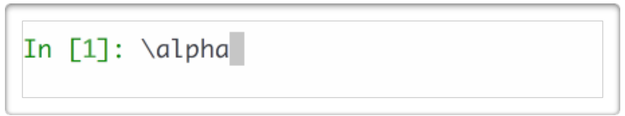
- Read-Eval-Print Loop:
  - Read the code
  - Evaluate it
  - Print the results
  - Repeat

IPython vs Python REPL



Features

- **Syntax highlighting**
- **Tab completion**:
  - keywords, modules, methods, variables
  - files in the current directory
  - unicode characters!



- **Smart indentation**
- **History search**:
  - ↑ or ↓
  - text + ↑ or ↓
  - **Ctrl+R** + text + ↑ or ↓

FEATURES !!!


%psource
magics
profiles
shortcuts
input_transformation
verbosity
kernels
autocalls
hooks
introspection
%edit
debugging
configurable
unicode
extensions
%%javascript
%store
events
caching
magics
asynchronous
shell
profiles
macros
storage
%run
%recall
doctest
configurable
verbosity
input_transformation
startup_files
profiling
autoquotes
kernels
debugging
shortcuts
asynchronous
%history
%timeit
%%bash
%pfile
%debug
%edit

Dynamic object introspection

Need information about classes, variables, functions or modules?

a_variable? or **?a_variable**

```
In [2]: os?
Type:        module
String form: <module 'os' from '/Users/switowski/.pyenv/versions/3.7.2/lib/python3.7/os.py'>
File:        ~/.pyenv/versions/3.7.2/lib/python3.7/os.py
Docstring:
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix or nt, e.g. unlink, stat, etc.
  - os.path is either posixpath or ntpath
  - os.name is either 'posix' or 'nt'
  - os.curdir is a string representing the current directory (always '.')
  - os.pardir is a string representing the parent directory (always '..')
  - os.sep is the (or a most common) pathname separator ('/' or '\\')
  - os.extsep is the extension separator (always '.')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).

In [3]:
```

Dynamic object introspection

Need **more** information?

a_variable?? or **??a_variable**

Dynamic object introspection

Forgot the name of a function?

Use * to list all functions matching a string

```
In [1]: import os

In [2]: os.*dir*?
os.__dir__
os.chdir
os.curdir
os.fchdir
os.listdir
os.makedirs
os.mkdir
os.pardir
os.removedirs
os.rmdir
os.scandir
os.supports_dir_fd

In [3]:
```

Input and output caching

- IPython stores the **input** and **output** of each command in the **current session**
- It also stores the **input** (and **output** - if enabled in the settings) of the **previous sessions**

Input caching

Input commands are stored in:

- (*for the last 3 inputs)* `_i`, `_ii`, `_iii`
- `_i`<cell_number>
- `_ih`[<cell_number>]
- `In`[<cell_number>]

`_ih` and **`In`** are lists indexed from 1!

```
In [9]: 1+2
Out[9]: 3

In [10]: _i
Out[10]: '1+2'

In [11]: _i9
Out[11]: '1+2'

In [12]: _ih[9]
Out[12]: '1+2'

In [13]: In[9]
Out[13]: '1+2'

In [14]:
```

Output caching

Output commands are stored in:

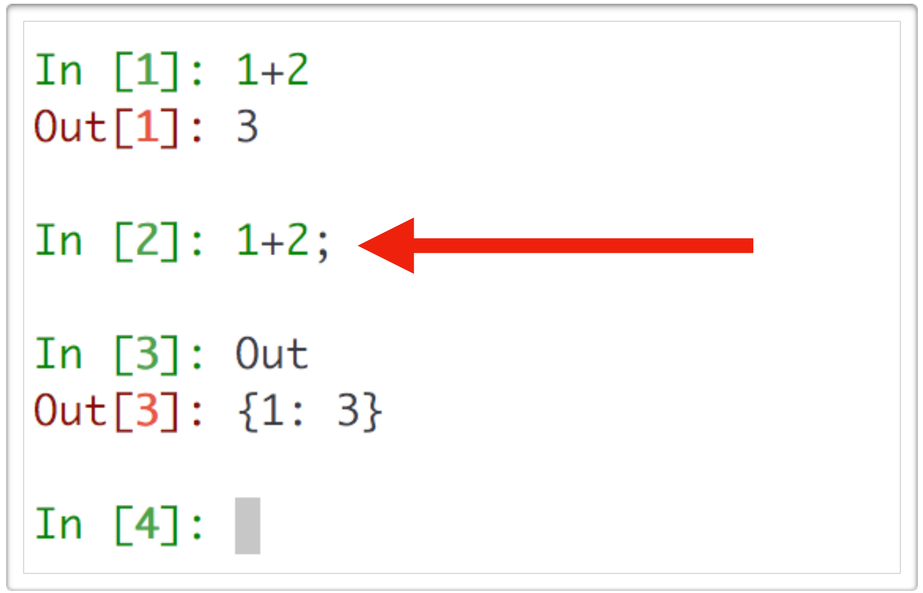
- (*for the last 3 outputs)* _, __, ___
- _<cell_number>
- _oh[<cell_number>]
- Out[<cell_number>]

```
In [9]: 3 + 0.14
Out[9]: 3.14

In [10]: _
Out[10]: 3.14

In [11]: _9
Out[11]: 3.14

In [12]: _oh[9]
Out[12]: 3.14

In [13]: Out[9]
Out[13]: 3.14

In [14]:
```

Why caching matters?

- Did you ever run a command that returns a value just to realize later that you want to do something with that value?
- And maybe it’s a very slow command or you can’t rerun it (authentication expired)
- With IPython you can just retrieve the output from the cache!

Suppressing the output



Magic functions

- Magic functions - helper functions that starts with **%** or **%%**, e.g:

```
%history -n -o 1-10
```

- IPython magic functions **!=** Python magic methods (__add__)!

% vs %%

- `%timeit` is a **line magic function** (similar to shell commands)

```
In [6]: %timeit -n 100 -r 3 sum(range(10000))
198 µs ± 17.9 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
```

% vs %%

- `%%timeit` is a **cell magic function**

```
# Measure the inefficient way to sum the elements
In [12]: %%timeit -n 100 -r 3
    ...: total = 0
    ...: for x in range(10000):
    ...:     total += x
    ...:
534 µs ± 13.7 µs per loop (mean ± std. dev. of 3 runs, 100 loops each)
```

124 magic functions of IPython

```
In [2]: %lsmagic
Out[2]:
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %autoindent  %automagic  %bookmark  %cat  %cd  %clear
%colors  %conda  %config  %cp  %cpaste  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env  %gui
%hist  %history  %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  %pip  %popd  %pprint
%precision  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %quickref  %recall  %rehashx
%reload_ext  %rep  %rerun  %reset  %reset_selective  %rm  %rmdir  %run  %save  %sc  %set_env  %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  %%js  %%latex  %%markdown
%%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.
```

My favorite magic functions

%alias

%cpaste

%debug

%edit

%history

%load

%load_ext

%ls

%macro

%prun

%recall

%rehashx

%rerun

%save

%store

%timeit

%who / %whos

%xmode

My favorite magic functions

%history

%edit

%run

%rerun

%recall

%macro

%save

%pastebin

%store

%who / %whos

%history

Prints the input history:

```
%history

%history 5

%history 2-3 5 7-9
```

```
In [1]: 1+2
Out[1]: 3

In [2]: print('hello world')
hello world

In [3]: 2+5
Out[3]: 7

In [4]: sum(range(1000))
Out[4]: 499500

In [5]: %history
1+2
print('hello world')
2+5
sum(range(1000))
%history

In [6]: %history 4
sum(range(1000))

In [7]: %history 1-2 4
1+2
print('hello world')
sum(range(1000))
```

%history

Prints the input history:

```
%history

%history 5

%history 2-3 5 7-9
```

```
In [1]: 1+2
Out[1]: 3

In [2]: print('hello world')
hello world

In [3]: 2+5
Out[3]: 7

In [4]: sum(range(1000))
Out[4]: 499500

In [5]: %history
1+2
print('hello world')
2+5
sum(range(1000))
%history

In [6]: %history 4
sum(range(1000))

In [7]: %history 1-2 4
1+2
print('hello world')
sum(range(1000))
```

range in IPython

- `%history 2-3 5 7-9`
  - Range 7-9 means: line 7,8 AND 9 (unlike Python's *range*)
  - You can mix ranges and single lines (duplicates are fine too!)
- `%history 457/7` # Line 7 from session number 457
- `%history ~2/7` # Line 7 from 2 sessions ago
- `%history ~1/` # The whole previous session
- `%history ~8/1-~6/5` # From the 1st line 8 sessions ago until the 5th line of 6 sessions ago

%edit

Opens a temporary file (in your favorite editor*.) and executes the code after you save and quit:

```
%edit
```

```
%edit -p
```

<*F2*> is a shortcut for `%edit`

** Based on the $EDITOR (or $VISUAL) environment variable. By default uses **vim**, **nano** or **notepad**.*

%edit **ARGUMENT**

Where argument can be:

- a filename
- range of input history
- a variable
- an object (e.g. a function)
- a macro

%run

- Run a Python script and load its data into the current namespace
- Useful when writing a module (instead of `importlib.reload()`)
- Bonus:
  - `%autoreload` - **always** reload a module before executing a function

Other magic functions

- `%rerun` - rerun a command from the history
- `%recall` - like %rerun, but let's you edit the commands before executing
- `%macro` - store previous commands as a macro
- `%save` - save commands to a file
- `%pastebin` - save commands to a pastebin (similar to GitHub gist)
- `%store` - save macros, variables or aliases in IPython storage
- `%who` and `%whos` - print all interactive variables

Cell magics for different programming languages

%%python2

%%bash

%%ruby

%%javascript

```
In [1]: print "this" "won't" "work"

  File "<ipython-input-1-94cbffc45fdb>", line 1
    print "this" "won't" "work"
                ^
SyntaxError: Missing parentheses in call to 'print'. Did
 you mean print("this" "won't" "work")?


In [2]: %%python2
   ...: print "but" "this" "will"
   ...:
   ...:

butthiswill

In [3]: %%ruby
   ...: puts "hello from Ruby!"
   ...:
   ...:

hello from Ruby!

In [4]:
```

Writing magic functions

How to write a magic function:

1. Write a function

2. Decorate it with `@register_line_magic` or `@register_cell_magic`

Writing magic functions

Reverse a string:

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

```
In [2]: %reverse hello world
Out[2]: 'dlrow olleh'
```

Writing magic functions

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

```
In [2]: %reverse hello world
Out[2]: 'dlrow olleh'
```

Writing magic functions

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

```
In [2]: %reverse hello world
Out[2]: 'dlrow olleh'
```

Writing magic functions

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

```
In [2]: %reverse hello world
Out[2]: 'dlrow olleh'
```

Writing magic functions

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

```
In [2]: %reverse hello world
Out[2]: 'dlrow olleh'
```

Writing magic functions

More information on magic functions:

- IPython documentation
- Cell magic function that runs mypy

Extensions

- Extensions - an easy way to make your magic functions reusable and share them with others through PyPI...
- ... but they not only limited to magic functions (key bindings, custom colors, custom IPython configuration, etc.)

Writing an extension

- To create an extension you need to create a file containing `load_ipython_extension` function (and optionally the `unload_ipython_extension`)

```python
# myextension.py

def load_ipython_extension(ipython):
    # The `ipython` argument is the currently active `InteractiveShell`
    # instance, which can be used in any way. This allows you to register
    # new magics or aliases, for example.

def unload_ipython_extension(ipython):
    # If you want your extension to be unloadable, put that logic here.
```

https://ipython.readthedocs.io/en/stable/config/extensions/index.html

- And save the file in a folder called `.ipython/extensions`

Writing an extension

Let’s turn our magic function into an extension!

Writing an extension

```
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]
```

Writing an extension

```
from IPython.core.magic import register_line_magic


def load_ipython_extension(ipython):
    @register_line_magic("reverse")
    def lmagic(line):
        "Line magic to reverse a string"
        return line[::-1]
```

Writing an extension

```
# ~/.ipython/extensions/reverser.py


from IPython.core.magic import register_line_magic


def load_ipython_extension(ipython):
    @register_line_magic("reverse")
    def lmagic(line):
        "Line magic to reverse a string"
        return line[::-1]
```

Writing an extension

```
In [1]: %load_ext reverser
Loading extensions from ~/.ipython/extensions is deprecated. We recommend managing extensions
like any other Python packages, in site-packages.

In [2]: %reverse Hello world!
Out[2]: '!dlrow olleH'

In [3]:
```

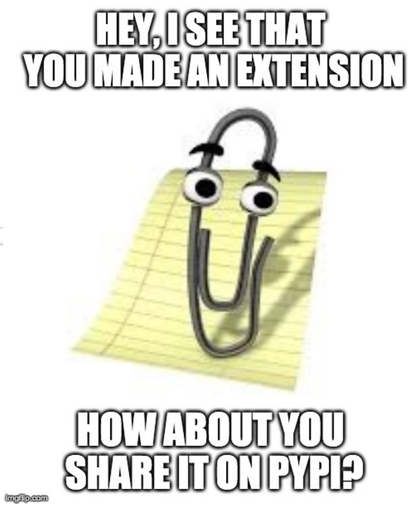
Writing an extension

```
In [1]: %load_ext reverser
Loading extensions from ~/.ipython/extensions is deprecated. We recommend managing extensions
like any other Python packages, in site-packages.

In [2]: %reverse Hello world!
Out[2]: '!dlrow olleH'

In [3]:
```

```python
# ~/.ipython/extensions/reverser.py


from IPython.core.magic import register_line_magic


def load_ipython_extension(ipython):
    @register_line_magic("reverse")
    def lmagic(line):
        "Line magic to reverse a string"
        return line[::-1]
```

Writing an extension

```
In [1]: %load_ext reverser
Loading extensions from ~/.ipython/extensions is deprecated. We recommend managing extensions
like any other Python packages, in site-packages.

In [2]: %reverse Hello world!
Out[2]: '!dlrow olleH'

In [3]:
```



Deprecation warning discussed here and here

Publishing extension on PyPI

Let’s publish my little extension on PyPI:

https://pypi.org/project/IPythonReverser

You can now install it with:

```
pip install IPythonReverser
```

Load in IPython with:

```
%load_ext ipython_reverser
```

And run:

```
%reverse Hello world
```

Where to find extensions?

- Extensions Index - a wiki page in IPython repository (some extensions are *old*!)
- Framework::IPython filter on PyPI - the recommended way to share extensions
- Search for “IPython” or “IPython magic” on PyPI

Extensions - examples

- IPython-SQL - interact with SQL databases from IPython
- IPython Cypher - interact with Neo4j
- Django ORM magic - define Django models on the fly

Shell commands

- Commands starting with **!** are treated as shell commands
- Some common commands don’t require **!** prefix (cd, ls, pwd, etc.)

```
In [1]: cd test_dir/
/Users/switowski/workspace/test_dir

In [2]: ls
test_file

In [3]: !echo "hello world" > new_file

In [4]: !cat new_file
hello world

In [5]:
```

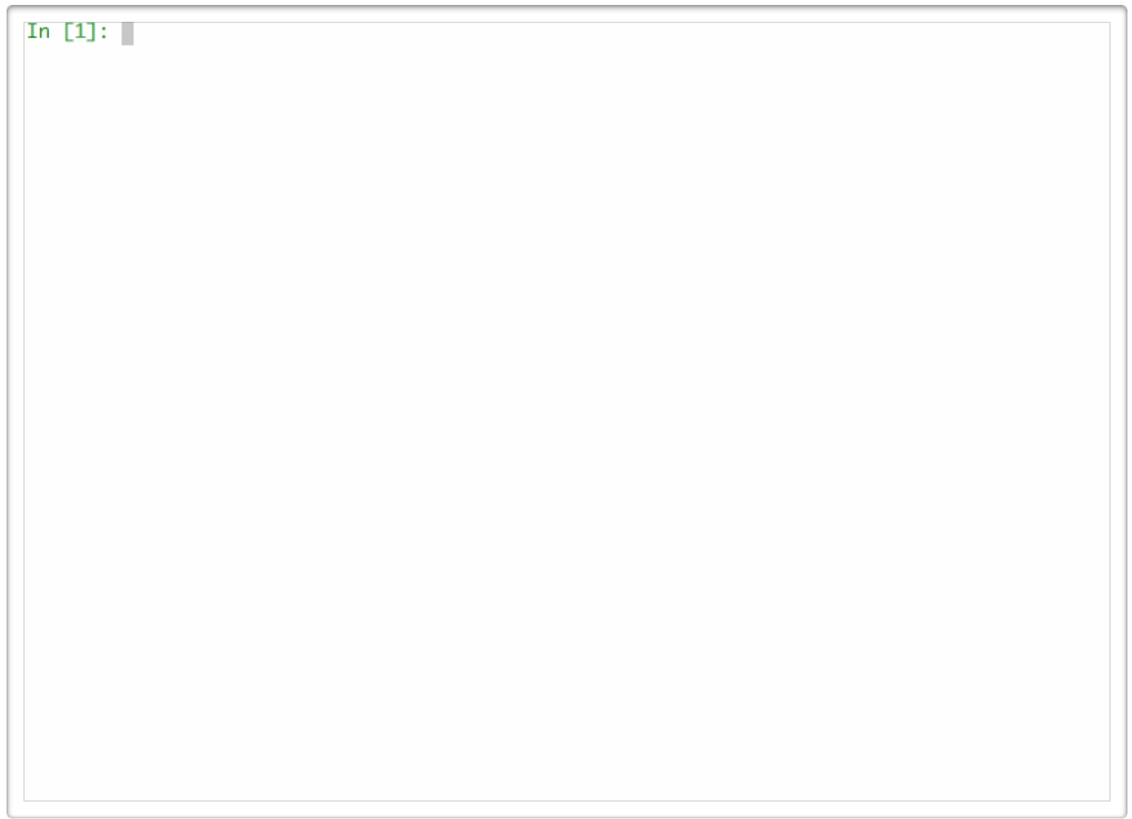
%alias

Similar to Linux alias command, they let you call a **system command** under a different name:

```
In [1]: %alias lr ls -alrt

In [2]: lr
total 8
drwxr-xr-x  31 switowski  staff  992 May 26 20:14 ..
-rw-r--r--   1 switowski  staff    0 May 26 20:16 test_file
drwxr-xr-x   4 switowski  staff  128 May 26 20:17 .
-rw-r--r--   1 switowski  staff   12 May 26 20:17 new_file

In [3]: %alias print echo %s %s

In [4]: %print hello world
hello world
```

%rehashx

Loads all executables from $PATH into the alias table



%xmode

Changes how verbose the exceptions should be

```
In [10]: %xmode minimal
Exception reporting mode: Minimal

In [11]: function1()
IndexError: list index out of range
```

%xmode

Changes how verbose the exceptions should be

```
In [14]: %xmode plain
Exception reporting mode: Plain

In [15]: function1()
Traceback (most recent call last):
  File "<ipython-input-15-c0b3cafe2087>", line 1, in <module>
    function1()
  File "/Users/switowski/workspace/playground/my_broken_function.py"
    return function2(5)
  File "/Users/switowski/workspace/playground/my_broken_function.py"
    total += a_list[x]
IndexError: list index out of range
```

%xmode

Changes how verbose the exceptions should be

```
In [18]: %xmode context
Exception reporting mode: Context

In [19]: function1()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-19-c0b3cafe2087> in <module>
----> 1 function1()

~/workspace/playground/my_broken_function.py in function1()
      1 def function1():
----> 2     return function2(5)
      3
      4 def function2(param):
      5     a_list = [1,2,3,4]

~/workspace/playground/my_broken_function.py in function2(param)
      6     total = 0
      7     for x in range(param):
----> 8         total += a_list[x]
      9     return total
     10

IndexError: list index out of range
```

%xmode

Changes how verbose the exceptions should be

```
In [20]: %xmode verbose
Exception reporting mode: Verbose

In [21]: function1()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-c0b3cafe2087> in <module>
----> 1 function1()
        global function1 = <function function1 at 0x10df42158>

~/workspace/playground/my_broken_function.py in function1()
      1 def function1():
----> 2     return function2(5)
        global function2 = <function function2 at 0x10df2c6a8>
      3
      4 def function2(param):
      5     a_list = [1,2,3,4]

~/workspace/playground/my_broken_function.py in function2(param=5)
      6     total = 0
      7     for x in range(param):
----> 8         total += a_list[x]
        total = 10
        a_list = [1, 2, 3, 4]
        x = 4
      9     return total
     10

IndexError: list index out of range
```

Autoawait

Asynchronous code in REPL

```
❯ ipython
Python 3.7.2 (default, Jan 25 2019, 18:07:26)
Type 'copyright', 'credits' or 'license' for more informa
tion
IPython 7.4.0 -- An enhanced Interactive Python. Type '?'
 for help.

In [1]: import aiohttp

In [2]: session = aiohttp.ClientSession()
/Users/switowski/.virtualenvs/testipython/bin/ipython:1:
DeprecationWarning: The object should be created from asy
nc function
  #!/Users/switowski/.virtualenvs/testipython/bin/python

In [3]: result = session.get("https://api.github.com")

In [4]: response = await result

In [5]: response
Out[5]:
<ClientResponse(https://api.github.com/) [200 OK]>
<CIMultiDictProxy('Server': 'GitHub.com', 'Date': 'Mon, 2
7 May 2019 13:20:45 GMT', 'Content-Type': 'application/js
on; charset=utf-8', 'Transfer-Encoding': 'chunked', 'Stat
us': '200 OK', 'X-RateLimit-Limit': '60', 'X-RateLimit-Re
maining': '59', 'X-RateLimit-Reset': '1558966845', 'Cache
```

```
❯ python
Python 3.7.2 (default, Jan 25 2019, 18:07:26)
[Clang 10.0.0 (clang-1000.10.44.4)] on darwin
Type "help", "copyright", "credits" or "license" for more
 information.
>>> import aiohttp
>>> session = aiohttp.ClientSession()
__main__:1: DeprecationWarning: The object should be crea
ted from async function
>>> result = session.get("https://api.github.com")
>>> response = await result
  File "<stdin>", line 1
SyntaxError: 'await' outside function
>>>
```

This is NOT a valid Python code!
Don't do this in production!

Demo mode

```
# demo.py

print('Hello, welcome to an interactive IPython demo.')
# <demo> --- stop ---

x = 1
y = 2

# <demo> --- stop ---

z = x+y

print('z=',x)

# <demo> --- stop ---
print('z is now:', z)

print('bye!')
```

```
from IPython.lib.demo import Demo

mydemo = Demo("demo.py")
mydemo()
```

Demo mode


In [1]: from IPython.lib.demo import Demo
In [2]: mydemo = Demo("demo.py")
In [3]:

Configuration

- IPython has pretty good defaults
- But if you need to change something, there is a configuration file:

```
~/.ipython/profile_default/ipython_config.py
```

- To create this file, run:

```
ipython profile create
```

```
# ipython_config.py

# Configuration file for ipython.

#------------------------------------------------------------------------------
# InteractiveShellApp(Configurable) configuration
#------------------------------------------------------------------------------

## Execute the given command string.
#c.InteractiveShellApp.code_to_run = ''

## Run the file referenced by the PYTHONSTARTUP environment variable at IPython
#  startup.
#c.InteractiveShellApp.exec_PYTHONSTARTUP = True

## List of files to run at IPython startup.
#c.InteractiveShellApp.exec_files = []

## lines of code to run at IPython startup.
#c.InteractiveShellApp.exec_lines = []

## A list of dotted module names of IPython extensions to load.
#c.InteractiveShellApp.extensions = []

## dotted module name of an IPython extension to load.
#c.InteractiveShellApp.extra_extension = ‘'

(…)
```

In `ipython_config.py` you can:

- execute specific lines of code at startup
- execute files at startup
- load extensions
- disable the banner and configuration files (faster startup)
- disable/enable autocalls
- change the color schema
- change the size of output cache or history length
- automatically start pdb after each exception
- change exception mode
- select editor for the %edit
- set the SQLite DB location
- enable output caching between sessions
- restore all variables from %store on startup

```
~/.ipython/profile_default
❯ ls -al
total 944
drwxr-xr-x  10 switowski  staff     320 May 27 08:51 .
drwxr-xr-x   7 switowski  staff     224 Apr 13 08:25 ..
drwxr-xr-x   7 switowski  staff     224 Apr 10 13:29 db
-rw-r--r--   1 switowski  staff  442368 May 27 08:51 history.sqlite
-rw-r--r--   1 switowski  staff   23668 Apr  8 10:35 ipython_config.py
drwxr-xr-x   2 switowski  staff      64 May  7  2018 log
drwx------   2 switowski  staff      64 May  7  2018 pid
drwx------   2 switowski  staff      64 May  7  2018 security
drwxr-xr-x   4 switowski  staff     128 May 22 07:05 startup
drwxr-xr-x   3 switowski  staff      96 Apr 13 08:25 static
```

```
~/.ipython/profile_default
❯ ls -al
total 944
drwxr-xr-x  10 switowski  staff     320 May 27 08:51 .
drwxr-xr-x   7 switowski  staff     224 Apr 13 08:25 ..
drwxr-xr-x   7 switowski  staff     224 Apr 10 13:29 db
-rw-r--r--   1 switowski  staff  442368 May 27 08:51 history.sqlite
-rw-r--r--   1 switowski  staff   23668 Apr  8 10:35 ipython_config.py
drwxr-xr-x   2 switowski  staff      64 May  7  2018 log
drwx------   2 switowski  staff      64 May  7  2018 pid
drwx------   2 switowski  staff      64 May  7  2018 security
drwxr-xr-x   4 switowski  staff     128 May 22 07:05 startup
drwxr-xr-x   3 switowski  staff      96 Apr 13 08:25 static
```

Startup files

```
~/.ipython/profile_default/startup
❯ ls -al
total 8
drwxr-xr-x   3 switowski  staff   96 May 27 08:56 .
drwxr-xr-x  11 switowski  staff  352 May 27 08:56 ..
-rw-r--r--   1 switowski  staff  371 May  7  2018 README
~/.ipython/profile_default/startup
❯ cat README
This is the IPython startup directory

.py and .ipy files in this directory will be run *prior* to any code or files specified
via the exec_lines or exec_files configurables whenever you load this profile.

Files will be run in lexicographical order, so you can control the execution order of files
with a prefix, e.g.::

    00-first.py
    50-middle.py
    99-last.ipy
```

Startup files

```
~/.ipython/profile_default/startup
❯ ls -al
total 16
drwxr-xr-x   4 switowski  staff  128 May 27 09:01 .
drwxr-xr-x  11 switowski  staff  352 May 27 09:02 ..
-rw-r--r--   1 switowski  staff  371 May  7  2018 README
-rw-r--r--   1 switowski  staff  162 May 27 09:01 my_magic.py
~/.ipython/profile_default/startup
❯ cat my_magic.py
from IPython.core.magic import register_line_magic

@register_line_magic("reverse")
def lmagic(line):
    "Line magic to reverse a string"
    return line[::-1]

~/.ipython/profile_default/startup
❯ i

In [1]: %reverse Hello world!
Out[1]: '!dlrow olleH'
```

Startup files

- Large startup files == long IPython startup time!
- Use a separate profile instead

Profiles

- **Profiles** are like accounts on your computer (each has a separate configuration and startup files)
- Each profile is a separate directory in `.ipython` directory

Profiles

- Create a new profile:

```
$ ipython profile create foo
```

- Start IPython with that profile:

```
$ ipython --profile=foo
```

- By default, IPython starts with the *default* profile

Events

IPython.core.events.pre_execute()

Fires before code is executed in response to user/frontend action.

This includes comm and widget messages and silent execution, as well as user code cells.

IPython.core.events.pre_run_cell(*info*)

Fires before user-entered code runs.

Parameters

info (`ExecutionInfo`) – An object containing information used for the code execution.

IPython.core.events.post_execute()

Fires after code is executed in response to user/frontend action.

This includes comm and widget messages and silent execution, as well as user code cells.

IPython.core.events.post_run_cell(*result*)

Fires after user-entered code runs.

Parameters

result (`ExecutionResult`) – The object which will be returned as the execution result.

IPython.core.events.shell_initialized(*ip*)

Fires after initialisation of `InteractiveShell` .

This is before extensions and startup scripts are loaded, so it can only be set by subclassing.

Parameters

ip (`InteractiveShell`) – The newly initialised shell.

Events

- To add a callback to an event:
  - Define your callback (check Module: core.event documentation)
  - Define `load_ipython_extension(ip)` function
    - Register callback with `ip.events.register()`
  - Load the extension (with `%load_ext` function)

Writing a custom event

To print all the variables after cell execution

```
class VarPrinter:
    def __init__(self, ip):
        self.ip = ip

    def post_run_cell(self, result):
        print("------------------------------")
        print("Variables after cell execution:")
        self.ip.run_line_magic("whos", '')


def load_ipython_extension(ip):
    vp = VarPrinter(ip)
    ip.events.register("post_run_cell", vp.post_run_cell)
```

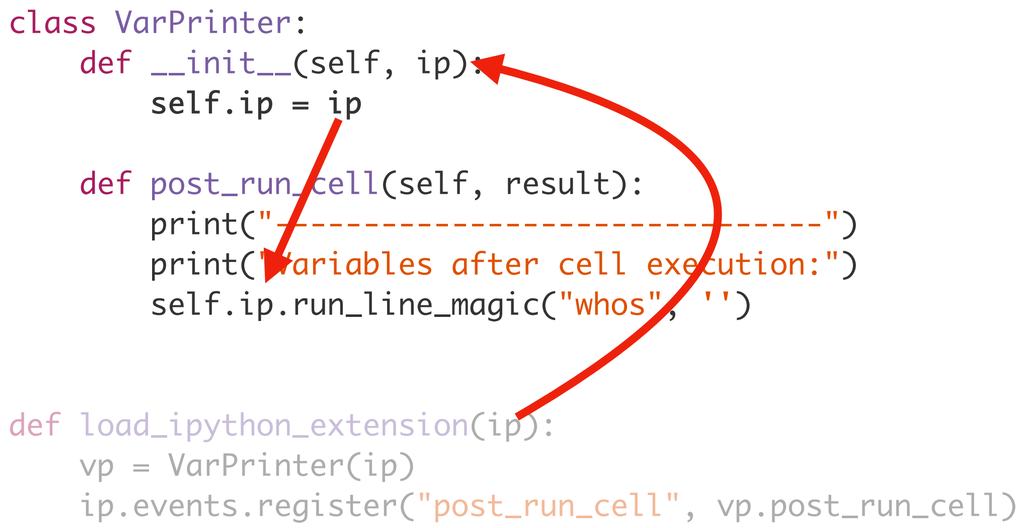
Writing a custom event

To print all the variables after cell execution

```
class VarPrinter:
    def __init__(self, ip):
        self.ip = ip

    def post_run_cell(self, result):
        print("------------------------------")
        print("Variables after cell execution:")
        self.ip.run_line_magic("whos", '')


def load_ipython_extension(ip):
    vp = VarPrinter(ip)
    ip.events.register("post_run_cell", vp.post_run_cell)
```

Writing a custom event

To print all the variables after cell execution



Writing a custom event

To print all the variables after cell execution

```
class VarPrinter:
    def __init__(self, ip):
        self.ip = ip

    def post_run_cell(self, result):
        print("------------------------------")
        print("Variables after cell execution:")
        self.ip.run_line_magic("whos", '')


def load_ipython_extension(ip):
    vp = VarPrinter(ip)
    ip.events.register("post_run_cell", vp.post_run_cell)
```

Writing a custom event

To print all the variables after cell execution

```
class VarPrinter:
    def __init__(self, ip):
        self.ip = ip

    def post_run_cell(self, result):
        print("------------------------------")
        print("Variables after cell execution:")
        # %whos would give a SyntaxError!
        self.ip.run_line_magic("whos", '')


def load_ipython_extension(ip):
    vp = VarPrinter(ip)
    ip.events.register("post_run_cell", vp.post_run_cell)
```

Writing a custom event

To print all the variables after cell execution

```
class VarPrinter:
    def __init__(self, ip):
        self.ip = ip

    def post_run_cell(self, result):
        print("------------------------------")
        print("Variables after cell execution:")
        self.ip.run_line_magic("whos", '')


def load_ipython_extension(ip):
    vp = VarPrinter(ip)
    ip.events.register("post_run_cell", vp.post_run_cell)
```

Writing a custom event

```
In [1]: %load_ext varprinter
Loading extensions from ~/.ipython/extensions is deprecated.
packages, in site-packages.
------------------------------
Variables after cell execution:
Interactive namespace is empty.

In [2]: a = 10
------------------------------
Variables after cell execution:
Variable   Type    Data/Info
----------------------------
a          int     10

In [3]: b = [1,2,3]
------------------------------
Variables after cell execution:
Variable   Type    Data/Info
----------------------------
a          int     10
b          list    n=3

In [4]:
```

Hooks

- Similar to events, used for example when:
  - Opening an editor (with `%edit`)
  - Shutting down IPython
  - Copying text from clipboard

Events vs Hooks

- There can be multiple callback functions run on one **event** (they are independent of each other)
- But only one function will run for a given **hook** (unless it fails - then the next function will be tried)!

Hooks

```
import os

def calljed(self, filename, linenum):
    "My editor hook calls the jed editor directly."
    print "Calling my own editor, jed ..."
    if os.system('jed +%d %s' % (linenum, filename)) != 0:
        raise TryNext()

def load_ipython_extension(ip):
    ip.set_hook('editor', calljed)
```

Example from the documentation

Hooks

```
import os

def calljed(self, filename, linenum):
    "My editor hook calls the jed editor directly."
    print "Calling my own editor, jed ..."
    if os.system('jed +%d %s' % (linenum, filename)) != 0:
        raise TryNext()

def load_ipython_extension(ip):
    ip.set_hook('editor', calljed)
```

Debugging

- IPython has been my default debugger since a long time (because of Sublime Text that I have used for years)

Debugging part 1:

Embedding

```
# embedding_example.py

a = 10
b = 15

from IPython import embed; embed()

print(f"a+b = {a+b}")
```

Debugging part 1:

Embedding

```python
# embedding_example.py

a = 10
b = 15

from IPython import embed; embed()

print(f"a+b = {a+b}")
```

```
› python ./embedding_example.py
Python 3.7.2 (default, Jan 25 2019, 18:07:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: a
Out[1]: 10

In [2]: b
Out[2]: 15

In [3]: a = 100

In [4]:
Do you really want to exit ([y]/n)? y

a+b = 115
```

Debugging part 2:

Debugger

`%run -d my_file.py`

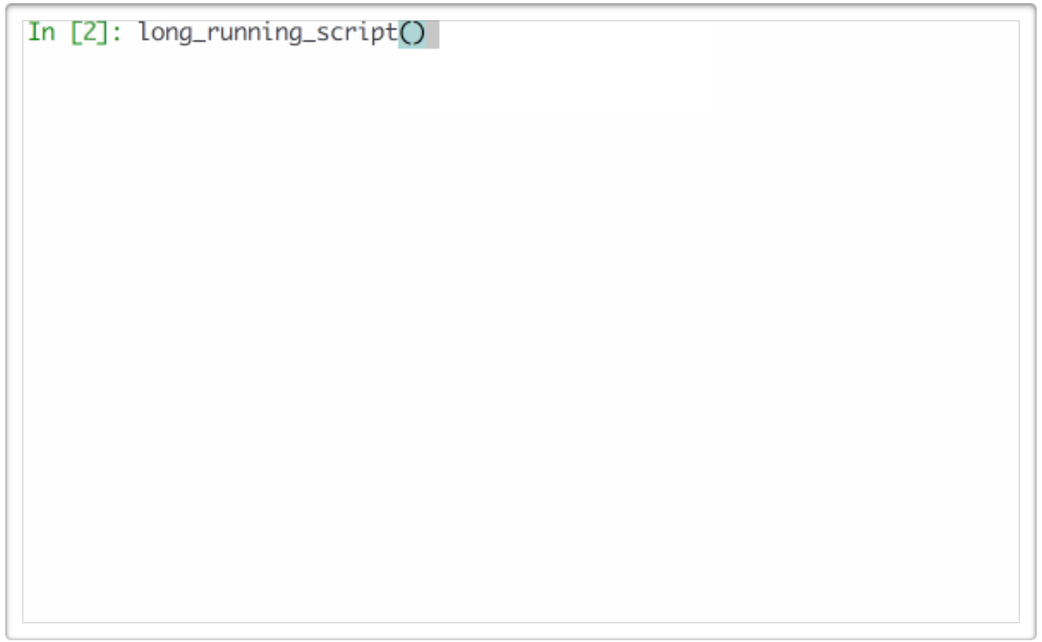
- Runs the file through pdb (ipdb)
- Puts the breakpoint on the 1st line

```
In [1]: %run -d myfile.py
Breakpoint 1 at /Users/switowski/workspace/playground/myfile.py:1
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> /Users/switowski/workspace/playground/myfile.py(1)<module>()
1---> 1 a = 10
      2 b = 15
      3
      4 print(f"a+b = {a+b}")

ipdb> next
> /Users/switowski/workspace/playground/myfile.py(2)<module>()
1     1 a = 10
----> 2 b = 15
      3
      4 print(f"a+b = {a+b}")

ipdb> continue
a+b = 25

In [2]:
```

Debugging part 3:

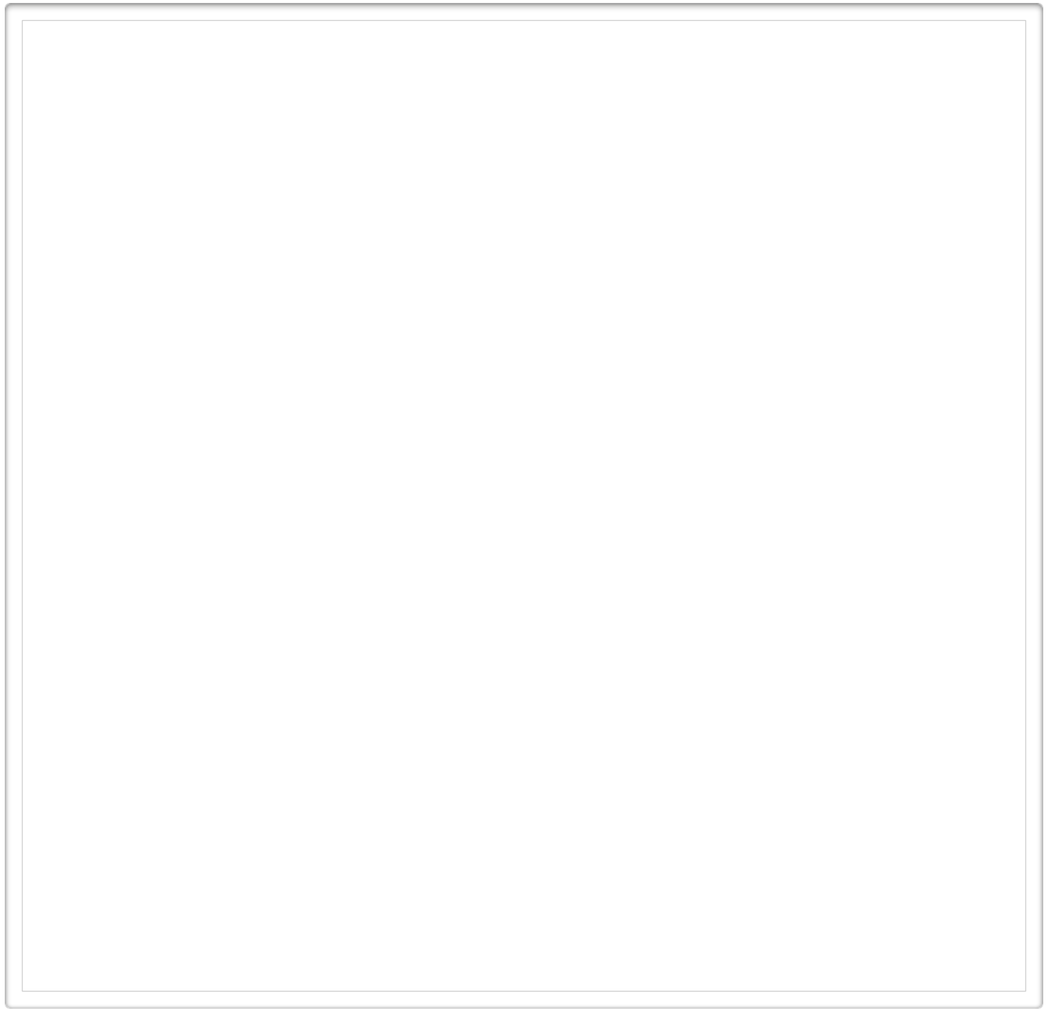
Post mortem debugger

Imagine you are running a Python script:



Debugging part 3:

Post mortem debugger



” I wish I ran this script with a debugger enabled!
Now I have to wait again to see what’s the problem 😭 *“*

-Me (and You?)

%debug to the rescue

```
     22
     23 def important_function(a):
---> 24     b = helper_function(a)
     25
     26 def helper_function(a):

~/workspace/playground/pmdebug.py in helper_function(a)
     26 def helper_function(a):
     27     b = a * 10
---> 28     c = a_method(b)
     29
     30 def a_method(a):

~/workspace/playground/pmdebug.py in a_method(a)
     31     b = 1000
     32     new_a = a - 980
---> 33     return do_calculations(new_a, b)
     34
     35 def do_calculations(a, b):

~/workspace/playground/pmdebug.py in do_calculations(a, b)
     34
     35 def do_calculations(a, b):
---> 36     return b / a
     37
     38 def long_running_script():

ZeroDivisionError: division by zero

In [3]:
```

Debugging part 4:

%pdb

```
In [1]: %pdb
Automatic pdb calling has been turned ON

In [2]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-9e1622b385b6> in <module>
----> 1 1/0

ZeroDivisionError: division by zero
> <ipython-input-2-9e1622b385b6>(1)<module>()
----> 1 1/0

ipdb>
```

Profiling

%time

Measure how long it takes to execute some code:

```
In [2]: %time run_calculations()
CPU times: user 2.68 s, sys: 10.9 ms, total: 2.69 s
Wall time: 2.71 s
Out[2]: 166616670000
```

%timeit

Measure how long it takes to execute some code.

But also figures out how many times it should run to give you reliable results:

```
In [5]: %timeit run_calculations()
2.82 s ± 124 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

%%timeit

```
In [1]: %%timeit [arguments] <optional_setup_code>
   ...: total = 0
   ...: for x in range(10000):
   ...:     for y in range(x):
   ...:         total += y
   ...:
2.7 s ± 25.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

%prun

```
In [1]: %prun a_slow_function()
         50035004 function calls in 12.653 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    10000    8.683    0.001   12.645    0.001 my_file.py:6(helper_function)
 49995000    3.956    0.000    3.956    0.000 my_file.py:15(check_factor)
    10000    0.005    0.000   12.650    0.001 my_file.py:1(important_function)
    10000    0.004    0.000    0.006    0.000 my_file.py:19(a_method)
        1    0.003    0.003   12.653   12.653 my_file.py:28(long_running_script)
    10000    0.001    0.000    0.001    0.000 my_file.py:24(do_calculations)
        1    0.000    0.000   12.653   12.653 {built-in method builtins.exec}
        1    0.000    0.000   12.653   12.653 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

line_profiler

- **%prun** returns a **function-by-function** report
- **%lprun** returns a **line-by-line** report
- It's not included by default in IPython:
  - Install from pip: `pip install line_profiler`
  - Load extension: `%load_ext line_profiler`

line_profiler

`%lprun` *-f function_name -f function2_name* **statement**

line_profiler

```
In [1]: %lprun -f long_running_script -f important_function long_running_script()
Timer unit: 1e-06 s

Total time: 27.3258 s
File: /Users/switowski/workspace/playground/my_file.py
Function: important_function at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def important_function(a, num):
     2     10000   27310547.0   2731.1     99.9      b = helper_function(a, num)
     3     10000      11686.0      1.2      0.0      b += 10
     4     10000       3560.0      0.4      0.0      return b

Total time: 27.3539 s
File: /Users/switowski/workspace/playground/my_file.py
Function: long_running_script at line 28

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    28                                           def long_running_script():
    29         1          2.0      2.0      0.0      total = 1
    30     10001       4033.0      0.4      0.0      for x in range(10000):
    31     10000   27349839.0   2735.0    100.0          total += important_function(total, x)
    32         1          0.0      0.0      0.0      return total
```

memory_profiler

- Profiles the memory usage of Python programs
- It’s not included by default in IPython:
  - Install from pip: `pip install memory_profiler`
  - Load extension: `%load_ext memory_profiler`

memory_profiler

`%mprun` *-f function_name -f function2_name* **statement**

memory_profiler

```
In [1]: %mprun -f memory_intensive memory_intensive()
Filename: /Users/switowski/workspace/playground/my_file.py

Line #    Mem usage    Increment   Line Contents
================================================
     1     57.4 MiB     57.4 MiB   def memory_intensive():
     2    820.3 MiB    762.9 MiB       a = [1] * (10 ** 8)
     3   2159.0 MiB   1338.6 MiB       b = [2] * (2 * 10 ** 8)
     4    618.1 MiB      0.0 MiB       del b
     5    618.1 MiB      0.0 MiB       return a
```

Kernels

- In IPython REPL, the “E” (Evaluation) happens in a separate process called **kernel**
- You can use a different kernel than the default (Python) one
    - The interface won’t change, but you will be using a different programming language (Ruby, JS, etc.)

How to change the kernel?

- Find a kernel you want
  (at Jupyter kernels wiki page)

Jupyter kernels

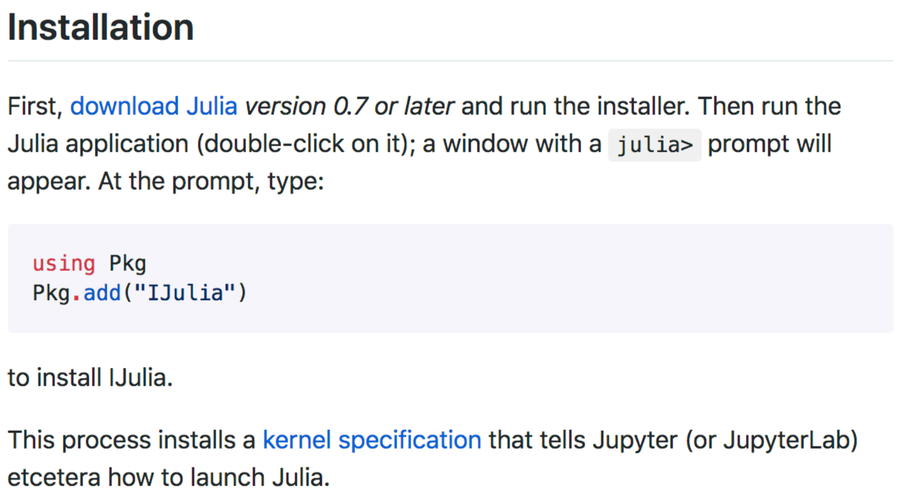
Kernel Zero is IPython, which you can get through ipykernel, and is still a dependency of jupyter. The IPython kernel can be thought of as a reference implementation, as CPython is for Python.

Here is a list of available kernels. If you are writing your own kernel, feel free to add it to the table!

Name	Jupyter/IPython Version	Language(s) Version	3rd party dependencies	Examp Notebo
Dyalog Jupyter Kernel		APL (Dyalog)	Dyalog >= 15.0	Noteboo
Coarray-Fortran	Jupyter 4.0	Fortran 2008/2015	GFortran >= 7.1, OpenCoarrays, MPICH >= 3.2	Demo, Binder demo
Ansible Jupyter Kernel	Jupyter 5.6.0.dev0	Ansible 2.x		Hello W
sparkmagic	Jupyter >=4.0	Pyspark (Python 2 & 3), Spark (Scala), SparkR (R)	Livy	Noteboo Docker Images
sas_kernel	Jupyter 4.0	python >= 3.3	SAS 9.4 or higher	
IPyKernel	Jupyter 4.0	python 2.7, >= 3.3	pyzmq	
IJulia		julia >= 0.3		
IHaskell		ghc >= 7.6		
IRuby		ruby >= 2.1		
IJavascript		nodejs >= 0.10		

How to change the kernel?

- Find a kernel you want
  (at Jupyter kernels wiki page)
- Install the dependencies and the kernel itself



Installation

First, download Julia *version 0.7 or later* and run the installer. Then run the Julia application (double-click on it); a window with a `julia>` prompt will appear. At the prompt, type:

```
using Pkg
Pkg.add("IJulia")
```

to install IJulia.

This process installs a kernel specification that tells Jupyter (or JupyterLab) etcetera how to launch Julia.

How to change the kernel?

- Find a kernel you want
  (at [Jupyter kernels wiki page]())
- Install the dependencies and the kernel itself
- Run it (either in IPython REPL or Jupyter Notebooks)

And if you really love IPython...

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)
- Or run commands like that:
  - ,print a b c # Equivalent to print(“a”, “b”, “c”)

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)
- Or run commands like that:
  - ,print a b c # Equivalent to print(“a”, “b”, “c”)
- Enable autoreloading, so you can change modules on the fly (no need to reimport them after changes)

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)
- Or run commands like that:
  - ,print a b c # Equivalent to print(“a”, “b”, “c”)
- Enable autoreloading, so you can change modules on the fly (no need to reimport them after changes)
- Turn on the “doctest mode” so you can easily write the doctest documentation

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)
- Or run commands like that:
  - ,print a b c # Equivalent to print(“a”, “b”, “c”)
- Enable autoreloading, so you can change modules on the fly (no need to reimport them after changes)
- Turn on the “doctest mode” so you can easily write the doctest documentation
- Turn IPython into your system shell (show current directory in prompt + autocalls + %rehashx)

You can:

- Enable autocalls, so you can skip brackets when calling functions (any  or  fans?)
- Or run commands like that:
  - ,print a b c # Equivalent to print(“a”, “b”, “c”)
- Enable autoreloading, so you can change modules on the fly (no need to reimport them after changes)
- Turn on the “doctest mode” so you can easily write the doctest documentation
- Turn IPython into your system shell (show current directory in prompt + autocalls + %rehashx)
- Add custom keyboard shortcuts
  - Or input transformations
  - Or AST transformations

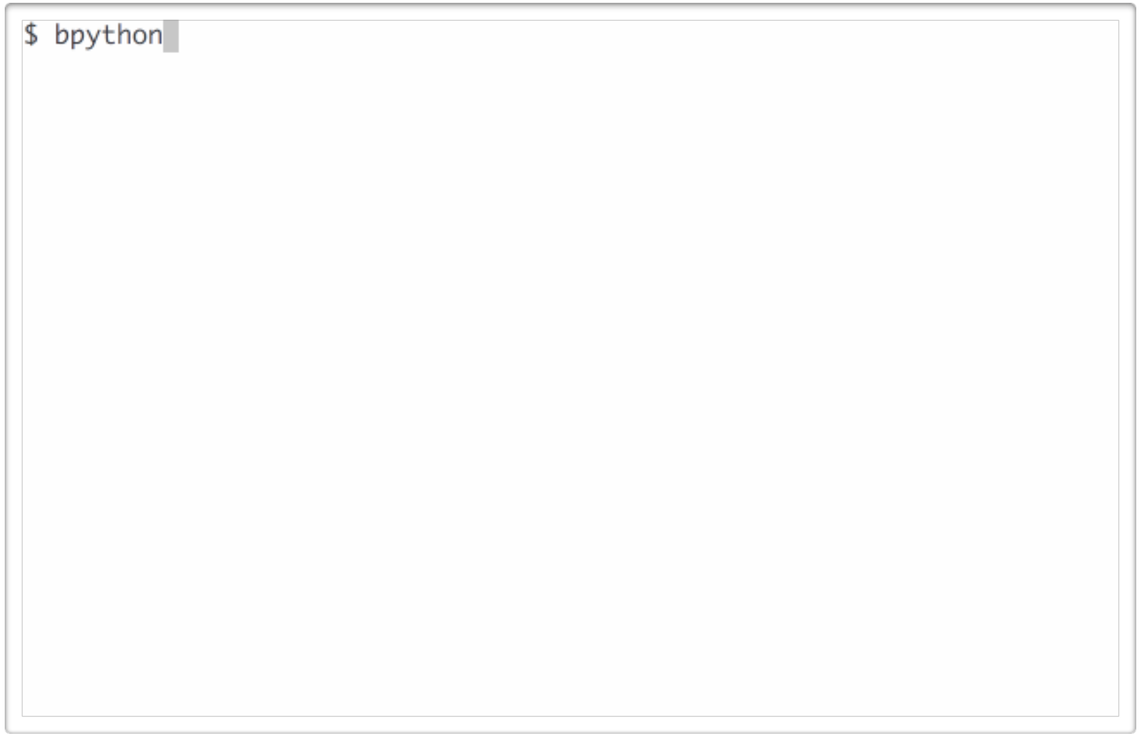
IPython alternatives

- bpython
- ptpython
- xonsh shell

bpython

Lightweight alternative to IPython:

- Syntax highlighting
- Smart indentation
- Autocompletion
- Suggestions when typing
- Rewind



https://bpython-interpreter.org

ptpython

- Syntax highlighting
- Multiline editing
- Autocompletion
- Shell commands
- Syntax validation
- Vim and Emacs mode
- Menus

https://pypi.org/project/ptpython/

xonsh shell

“Xonsh is a Python-powered, cross-platform, Unix-gazing shell language and command prompt. The language is a superset of Python 3.5+ with additional shell primitives that you are used to from Bash and IPython.”

– *https://xon.sh/index.html*

- Anthony Scopatz - xonsh - PyCon 2016
- Matthias Bussonnier, "Xonsh – put some Python in your Shell", PyBay2016

Thank you for listening!

And “thank you” creators of IPython for such an awesome tool!

Questions?

Slides:

bit.ly/advanced-ipython

@SebaWitowski