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
Don't try this at home!

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](https://www.youtube.com/watch?v=VQBZ2MqWBZI)
This talk is about IPython

But many of the things will apply to Jupyter as well
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 ↓
Dynamic object introspection

Need information about classes, variables, functions or modules?

`a_variable` or `?a_variable`
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) \(_i, _{ii}, _{iii}\)
- \(_i<\text{cell}\_\text{number}>\)
- \(_ih[<\text{cell}\_\text{number}>]\)
- \(\text{In}[<\text{cell}\_\text{number}>]\)

\(_ih\) and \(\text{In}\) are lists indexed from 1!
Output caching

Output commands are stored in:

- *(for the last 3 outputs) *_, __, ___*

- _<cell_number>

- _oh[<cell_number>]

- Out[<cell_number>]
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

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

In [2]: 1+2;

In [3]: Out
Out[3]: {1: 3}
```

In [4]:
Magic functions

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

  %history -n -o 1-10

• IPython magic functions != Python magic methods (`__add__`)!
• %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)
```
• `%%timeit` is a **cell magic function**

```python
# 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 %pin %pin %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

<table>
<thead>
<tr>
<th>Command 1</th>
<th>Command 2</th>
<th>Command 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>%alias</td>
<td>%load_ext</td>
<td>%rerun</td>
</tr>
<tr>
<td>%cpaste</td>
<td>%ls</td>
<td>%save</td>
</tr>
<tr>
<td>%debug</td>
<td>%macro</td>
<td>%store</td>
</tr>
<tr>
<td>%edit</td>
<td>%prun</td>
<td>%timeit</td>
</tr>
<tr>
<td>%history</td>
<td>%recall</td>
<td>%who / %whos</td>
</tr>
<tr>
<td>%load</td>
<td>%rehashx</td>
<td>%xmode</td>
</tr>
</tbody>
</table>
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
Prints the input history:

%history

%history 5

%history 2-3 5 7-9
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**.
Where argument can be:

- a filename
- range of input history
- a variable
- an object (e.g. a function)
- a macro
• 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
In [1]: print "this" "won't" "work"
```

File "<ipython-input-1-94cbbf45fdbc>", line 1
  print "this" "won't" "work"
^  SyntaxError: Missing parentheses in call to 'print'. Did you mean print("this" "won't" "work")?

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

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

```javascript
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:

```python
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'
```
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

```python
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

```python
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

```python
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.
```


- 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

```python
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

```python
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

```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]:
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](#)
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]: 
```
Similar to Linux alias command, they let you call a **system command** under a different name:
%rehashx

 Loads all executables from $PATH into the alias table
%xmode

Changes how verbose the exceptions should be

```python
In [10]: %xmode minimal
Exception reporting mode: Minimal
In [13]: function1()
IndexError: list index out of range
```
%xmode

Changes how verbose the exceptions should be
%xmode

Changes how verbose the exceptions should be
%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)
----> 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
    return total
10

IndexError: list index out of range
```

Autoawait

Asynchronous code in REPL

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

```python
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]:
```
• 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`

```bash
ls -al
```

```
total 8
drw-r-xr-x  3 switowski staff  96 May 27 08:56 .
drw-r-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

```bash
~/.ipython/profile_default/startup

ls -al

```

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

```python
from IPython.core.magic import register_line_magic

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

```

```bash
~/.ipython/profile_default/startup

```

```python
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** 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

```python
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

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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Data/Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>10</td>
</tr>
</tbody>
</table>

In [3]: b = [1,2,3]

Variables after cell execution:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Data/Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>10</td>
</tr>
<tr>
<td>b</td>
<td>list</td>
<td></td>
</tr>
</tbody>
</table>

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)!
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
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

```python
# 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}")
```

```bash
$ 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
```
%run -d my_file.py

• Runs the file through pdb (ipdb)

• Puts the breakpoint on the 1st line
Imagine you are running a Python script:

```python
In [2]: long_running_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

```python
22 def important_function(a):
23     ---> 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

~/workspace/playground/pmdebug.py in a_method(a)
31     b = 1000
32     new_a = a - 980

~/workspace/playground/pmdebug.py in do_calculations(a, b)
34     return do_calculations(new_a, b)
35 def do_calculations(a, b):

~/workspace/playground/pmdebug.py in do_calculations(a, b)
34     return do_calculations(a, b)
35 def do_calculations(a, b):

36     ---> 37         return b / a
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
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
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)
```
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)
```python
In [1]: %prun a_slow_function()
50035004 function calls in 12.653 seconds

Ordered by: internal time

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>49995000</td>
<td>3.956</td>
<td>0.000</td>
<td>12.650</td>
<td>0.001</td>
<td>my_file.py:15(check_factor)</td>
</tr>
<tr>
<td>10000</td>
<td>8.683</td>
<td>0.001</td>
<td>12.645</td>
<td>0.001</td>
<td>my_file.py:6(helper_function)</td>
</tr>
<tr>
<td>10000</td>
<td>0.004</td>
<td>0.000</td>
<td>0.006</td>
<td>0.000</td>
<td>my_file.py:19(a_method)</td>
</tr>
<tr>
<td>1</td>
<td>0.003</td>
<td>0.003</td>
<td>12.653</td>
<td>12.653</td>
<td>my_file.py:28(long_running_script)</td>
</tr>
<tr>
<td>10000</td>
<td>0.001</td>
<td>0.000</td>
<td>0.001</td>
<td>0.000</td>
<td>my_file.py:24(do_calculations)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>12.653</td>
<td>12.653</td>
<td>{built-in method builtins.exec}</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>12.653</td>
<td>12.653</td>
<td>&lt;string&gt;:1(&lt;module&gt;)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>{method 'disable' of '_lsprof.Profiler' objects}</td>
</tr>
</tbody>
</table>
```
%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
%lprun -f function_name -f function2_name statement
```python
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

<table>
<thead>
<tr>
<th>Line</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>def important_function(a, num):</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>27310547.0</td>
<td>2731.1</td>
<td>99.9</td>
<td>b = helper_function(a, num)</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>11686.0</td>
<td>1.2</td>
<td>0.0</td>
<td>b += 10</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>3560.0</td>
<td>0.4</td>
<td>0.0</td>
<td>return b</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>def long_running_script():</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>total = 1</td>
</tr>
<tr>
<td>30</td>
<td>10001</td>
<td>4033.0</td>
<td>0.4</td>
<td>0.0</td>
<td>for x in range(10000):</td>
</tr>
<tr>
<td>31</td>
<td>10000</td>
<td>27349839.0</td>
<td>2735.0</td>
<td>100.0</td>
<td>total += important_function(total, x)</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>return total</td>
</tr>
</tbody>
</table>
```
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
In [1]: %mprun -f memory_intensive memory_intensive()
Filename: /Users/switowski/workspace/playground/my_file.py

<table>
<thead>
<tr>
<th>Line #</th>
<th>Mem usage</th>
<th>Increment</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.4 MiB</td>
<td>57.4 MiB</td>
<td>def memory_intensive():</td>
</tr>
<tr>
<td>2</td>
<td>820.3 MiB</td>
<td>762.9 MiB</td>
<td>a = [1] * (10 ** 8)</td>
</tr>
<tr>
<td>3</td>
<td>2159.0 MiB</td>
<td>1338.6 MiB</td>
<td>b = [2] * (2 * 10 ** 8)</td>
</tr>
<tr>
<td>4</td>
<td>618.1 MiB</td>
<td>0.0 MiB</td>
<td>del b</td>
</tr>
<tr>
<td>5</td>
<td>618.1 MiB</td>
<td>0.0 MiB</td>
<td>return a</td>
</tr>
</tbody>
</table>
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](#))
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:

```plaintext
using Pkg
Pkg.add("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.”


- Anthony Scopatz - [xonsh](http://xon.sh) - PyCon 2016

- Matthias Bussonnier, "[Xonsh – put some Python in your Shell](http://xon.sh/index.html)", PyBay2016
Thank you for listening!

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

Slides:

bit.ly/advanced-ipython

@SebaWitowski