magic

December 27, 2019

0.1 Magic

0.2 BASIC MAGICS

There are two categories of magic: line magics and cell magics. Respectively, they act on a single line or can be spread across multiple lines or entire cells. To see the available magics, you can do the following:

[21]:
```
%lsmagic
```

[21]:
```
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark
%cat  %cd  %clear  %colors  %conda  %config  %connect_info  %cp  %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  %lprun  %ls  %lsmagic  %lx  %macro  %magic  %man
%matplotlib  %memit  %mkdir  %more  %mprun  %mv  %notebook  %page  %pastebin
%pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprint  %precision
%prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %qtconsole  %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  %%memit  %%mprun  %%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.
```

As you can see, there are loads! Most are listed in the official documentation, which is intended as a reference but can be somewhat obtuse in places. Line magics start with a percent character %, and cell magics start with two, %%.

It's worth noting that ! is really just a fancy magic syntax for shell commands, and as you may have noticed IPython provides magics in place of those shell commands that alter the state of the shell and are thus lost by !. Examples include %cd, %alias and %env.

0.3 Autosaving

First up, the %autosave magic let's you change how often your notebook will autosave to its checkpoint file.

[1]:
```
%autosave 60
```

```
Autosaving every 60 seconds
```

0.4 Displaying Matplotlib Plots

One of the most common line magics for data scientists is surely %matplotlib, which is of course for use with the most popular plotting libary for Python, Matplotlib.

[2]:
```
%matplotlib inline
```

Providing the inline argument instructs IPython to show Matplotlib plot images inline, within your cell outputs, enabling you to include charts inside your notebooks. Be sure to include this magic before you import Matplotlib, as it may not work if you do not; many import it at the start of their notebook, in the first code cell.

0.5 Latex Magic

[2]:
```
%%latex

$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$
```

$P(A \mid B) = \frac{P(B|A)P(A)}{P(B)}$

0.6 Fortran

[19]:
```
#!pip install cython fortran-magic
#!pip install fortran-magic
```

[4]:
```
%load_ext fortranmagic
```

```
/Users/erikapat/anaconda3/lib/python3.7/site-packages/fortranmagic.py:147:
UserWarning: get_ipython_cache_dir has moved to the IPython.paths module since
IPython 4.0.
  self._lib_dir = os.path.join(get_ipython_cache_dir(), 'fortran')
```

[12]:
```
%%fortran
subroutine compute_fortran(x, y, z)
    real, intent(in) :: x(:), y(:)
    real, intent(out) :: z(size(x, 1))

    z = sin(x + y)

end subroutine compute_fortran
```

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

```
UsageError: Cell magic `%%fortran` not found.
```

0.7 Jupyter Extensions

[6]:
```
#!pip install https://github.com/ipython-contrib/jupyter_contrib_nbextensions/
↪tarball/master
#!pip install jupyter_nbextensions_configurator
#!jupyter contrib nbextension install --user
#!jupyter nbextensions_configurator enable --user
```

0.8 Interactive Widgets

- https://nbviewer.jupyter.org/github/quantopian/ipython/blob/master/examples/Interactive%20Widgets/I
- https://github.com/quantopian/ipython/tree/master/examples/Interactive%20Widgets

0.9 HTML

[7]:
```
#!pip install folium
```

[8]:
```
import pandas as pd
import folium
from matplotlib.colors import Normalize, rgb2hex
import matplotlib.cm as cm
```

[9]:
```
data = pd.read_csv('http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/
↪all_day.csv')
norm = Normalize(data['mag'].min(), data['mag'].max())

map = folium.Map(location=[48, -102], zoom_start=3)
```

[10]:
```
for eq in data.iterrows():
    color = rgb2hex(cm.OrRd(norm(float(eq[1]['mag']))))
    folium.CircleMarker([eq[1]['latitude'], eq[1]['longitude']],
                        popup=eq[1]['place'],
                        radius=20000*float(eq[1]['mag']),
                        line_color=color,
                        fill_color=color).add_to(map)
```

[11]:
```
map.save('fig/earthquake.html')
```

[12]:
```
with open('fig/earthquake.html', 'r') as f:
    contents = f.read()
    contents = contents.replace("http://cdn.leafletjs.com/leaflet-0.5/", "//
↪cdnjs.cloudflare.com/ajax/libs/leaflet/0.7.7/")
```

```
    with open('fig/earthquake2.html', 'w') as f:
        f.writelines(contents)
```

[13]:
```
%%HTML
<iframe width="100%" height="350" src="https://app.dominodatalab.com/r00sj3/
↪jupyter/raw/latest/results/earthquake2.html?inline=true"></iframe>
```

```
<IPython.core.display.HTML object>
```

0.10 Timing and Profiling in IPython

- **%time & %timeit**: See how long a script takes to run (one time, or averaged over a bunch of runs).
- **%prun**: See how long it took each function in a script to run.
- **%lprun**: See how long it took each line in a function to run.
- **%mprun & %memit**: See how much memory a script uses (line-by-line, or averaged over a bunch of runs).

[14]:
```
!ipython --version
```

```
7.10.1
```

Most of the functionality we'll work with is included in the standard library, but if you're interested in line-by-line or memory profiling, go ahead and run through this setup. First, install the following:

[21]:
```
#!pip install line-profiler #does not work
!pip install git+https://github.com/rkern/line_profiler
```

```
Collecting git+https://github.com/rkern/line_profiler
  Cloning https://github.com/rkern/line_profiler to
/private/var/folders/nj/tcdmjt1954x5bp_7_l7cn2k40000gn/T/pip-req-build-vm7azbny
  Running command git clone -q https://github.com/rkern/line_profiler
/private/var/folders/nj/tcdmjt1954x5bp_7_l7cn2k40000gn/T/pip-req-build-vm7azbny
Requirement already satisfied: IPython>=0.13 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from line-
profiler==2.1.1) (7.10.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (2.0.9)
Requirement already satisfied: traitlets>=4.2 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (4.3.3)
Requirement already satisfied: jedi>=0.10 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (0.14.1)
Requirement already satisfied: pygments in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (2.5.2)
```

```
Requirement already satisfied: pexpect; sys_platform != "win32" in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (4.7.0)
Requirement already satisfied: decorator in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (4.4.1)
Requirement already satisfied: appnope; sys_platform == "darwin" in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (0.1.0)
Requirement already satisfied: setuptools>=18.5 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (42.0.2.post20191203)
Requirement already satisfied: pickleshare in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (0.7.5)
Requirement already satisfied: backcall in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from IPython>=0.13->line-
profiler==2.1.1) (0.1.0)
Requirement already satisfied: six>=1.9.0 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from prompt-
toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->IPython>=0.13->line-profiler==2.1.1)
(1.13.0)
Requirement already satisfied: wcwidth in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from prompt-
toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->IPython>=0.13->line-profiler==2.1.1)
(0.1.7)
Requirement already satisfied: ipython-genutils in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from
traitlets>=4.2->IPython>=0.13->line-profiler==2.1.1) (0.2.0)
Requirement already satisfied: parso>=0.5.0 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from
jedi>=0.10->IPython>=0.13->line-profiler==2.1.1) (0.5.2)
Requirement already satisfied: ptyprocess>=0.5 in
/Users/erikapat/anaconda3/lib/python3.7/site-packages (from pexpect;
sys_platform != "win32"->IPython>=0.13->line-profiler==2.1.1) (0.6.0)
Building wheels for collected packages: line-profiler
  Building wheel for line-profiler (setup.py) … done
  Created wheel for line-profiler:
filename=line_profiler-2.1.1-cp37-cp37m-macosx_10_9_x86_64.whl size=50311
sha256=509dcd9a854e4af5c27f21d910e1cd33491d0b33973acf4e2eb3694f91c7e74e
  Stored in directory:
/private/var/folders/nj/tcdmjt1954x5bp_7_l7cn2k40000gn/T/pip-ephem-wheel-cache-
boxnqmu7/wheels/b4/47/93/4e928668fc33778c7757bb92035808038f5e53dfc5dd9aee1a
Successfully built line-profiler
Installing collected packages: line-profiler
Successfully installed line-profiler-2.1.1
```

[15]:
```
#!pip install psutil
#!pip install memory_profiler
```

Next, create an IPython profile and extensions directory where we'll configure a couple of missing magic functions:

[16]:
```
!ipython profile create
```

See how long a script takes to run averaged over multiple runs.

[32]:
```
from past.builtins import xrange
```

[33]:
```
%time {1 for i in xrange(10*1000000)}
```

```
CPU times: user 286 ms, sys: 1.73 ms, total: 288 ms
Wall time: 287 ms
```

[33]:
```
{1}
```

It will limit the number of runs depending on how long the script takes to execute. Keep in mind that the timeit module in the standard library does not do this by default, so timing long running scripts that way may leave you waiting forever.

[6]:
```
%timeit 10*1000000
```

```
6.63 ns ± 0.145 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
```

[7]:
```
%timeit -n 1000 10*1000000
```

```
9.29 ns ± 0.143 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

%prun: See how long it took each function in a script to run.

[8]:
```
from time import sleep
```

[9]:
```
def foo(): sleep(1)
```

[10]:
```
def bar(): sleep(2)
```

[11]:
```
def baz(): foo(), bar()
```

[12]:
```
%prun baz()
```

```
         8 function calls in 3.005 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

```
     2    3.005    1.503    3.005    1.503 {built-in method time.sleep}
     1    0.000    0.000    3.005    3.005 {built-in method builtins.exec}
     1    0.000    0.000    3.005    3.005 <ipython-input-11-aeee66c4e941>:1(baz)
     1    0.000    0.000    2.005    2.005 <ipython-input-10-1428f9fb0b95>:1(bar)
     1    0.000    0.000    1.000    1.000 <ipython-input-9-a559e91038d2>:1(foo)
     1    0.000    0.000    3.005    3.005 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

%lprun: See how long it took each line in a function to run.

[1]:
```
from foo import foo
import sys
```

[2]:
```
%load_ext line_profiler
```

[3]:
```
%lprun -f foo foo(100000)
```

```
Timer unit: 1e-06 s

Total time: 0.101883 s
File: /Users/erikapat/Dropbox/PRUEBAS_DATA_SCIENCE/OOP/Python/foo.py
Function: foo at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           def foo(n):
     5         1       7197.0   7197.0      7.1      from past.builtins import xrange
     6         1          2.0      2.0      0.0      phrase = 'repeat me'
     7         1        464.0    464.0      0.5      pmul = phrase * n
     8         1      13340.0  13340.0     13.1      pjoi = ''.join([phrase for x in xrange(n)]
     9         1          1.0      1.0      0.0      pinc = ''
    10    100001      34145.0      0.3     33.5      for x in xrange(n):
    11    100000      46730.0      0.5     45.9          pinc += phrase
    12         1          4.0      4.0      0.0      del pmul, pjoi, pinc
```

0.11 Memory Profiling

See how much memory a script uses line by line. Let's take a look at the same foo() function that we profiled with %lprun - except this time we're interested in incremental memory usage and not execution time.

[7]:
```
%load_ext memory_profiler
```

```
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
```

[8]:
```
%mprun -f foo foo(100000)
```

```
Filename: /Users/erikapat/Dropbox/PRUEBAS_DATA_SCIENCE/OOP/Python/foo.py

Line #    Mem usage    Increment   Line Contents
================================================
     4     59.0 MiB     59.0 MiB   def foo(n):
     5     59.0 MiB      0.0 MiB       from past.builtins import xrange
     6     59.0 MiB      0.0 MiB       phrase = 'repeat me'
     7     59.0 MiB      0.0 MiB       pmul = phrase * n
     8     59.6 MiB      0.0 MiB       pjoi = ''.join([phrase for x in xrange(n)])
     9     59.6 MiB      0.0 MiB       pinc = ''
    10     62.9 MiB      0.0 MiB       for x in xrange(n):
    11     62.9 MiB      0.0 MiB           pinc += phrase
    12     60.4 MiB      0.0 MiB       del pmul, pjoi, pinc
```

%memit: See how much memory a script uses overall. %memit works a lot like %timeit except that the number of iterations is set with -r instead of -n.

[10]:
```
from past.builtins import xrange
%memit -r 3 [x for x in xrange(1000000)]
```

```
peak memory: 88.77 MiB, increment: 29.38 MiB
```

[11]:
```
!python -m trace --trace foo.py
```

```
 --- modulename: foo, funcname: <module>
foo.py(4): def foo(n):
```

0.12 Shell commands

[14]:
```
!echo Hello World!!
```

```
Hello World!!
```

[18]:
```
pip freeze | grep numpy
```

```
numpy==1.17.4
numpydoc==0.9.1
Note: you may need to restart the kernel to use updated packages.
```

0.13 Debugging

The more experienced reader may have had concerns over the ultimate efficacy of Jupyter Notebooks without access to a debugger. But fear not! The IPython kernel has its own interface to the Python debugger, pdb, and several options for debugging with it in your notebooks. Executing the %pdb line magic will toggle on/off the automatic triggering of pdb on error across all cells in your notebook.

[4]:
```
%pdb
```

```
Automatic pdb calling has been turned OFF
```

Another handy debugging magic is %debug, which you can execute after an exception has been raised to delve back into the call stack at the time of failure.

As an aside, also note how the traceback above demonstrates how magics are translated directly into Python commands, where %pdb became get_ipython().run_line_magic('pdb', ''). Executing this instead is identical to executing %pdb.

[1]:
```
x + 2
```

```
␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-1-2c7a9192c558> in <module>
   ----> 1 x + 2


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

[ ]:
```
%debug
```

```
> <ipython-input-1-2c7a9192c558>(1)<module>()
----> 1 x +
2
```

0.14 Configuring errors

Configuring Logging section

0.15 Macros

Like many users, you probably find yourself writing the same few tasks over and over again. Maybe there's a bunch of packages you always need to import when starting a new notebook, a few statistics that you find yourself computing for every single dataset, or some standard charts that you've produced countless times?

Jupyter lets you save code snippets as executable macros for use across all your notebooks. Although executing unknown code isn't necessarily going to be useful for anyone else trying to read or use your notebooks, it's definitely a handy productivity boost while you're prototyping, investigating, or just playing around.

Create and store a macro

[32]:
```
name = 'Tim'
print('Hello, %s!' % name)
```

```
Hello, Tim!
```

[34]:
```
%macro __hello_world 1
```

```
Macro `__hello_world` created. To execute, type its name (without quotes).
=== Macro contents: ===
name = 'Tim'
print('Hello, %s!' % name)
```

[35]:
```
%store __hello_world
```

```
Stored '__hello_world' (Macro)
```

Use a macro

[36]:
```
%store -r __hello_world
```

[37]:
```
__hello_world
```

```
Hello, Tim!
```

0.16 Transforming your jupyter notebook

```
jupyter nbconvert --to <format> notebook.ipynb
```

[ ]:
```

```

0.17 REFERENCES

- Read this: 1
- 2
- 3
- 4
- 5
- 6
- 7

[ ]:
```

```