

❗ Important

This documentation covers IPython versions 6.0 and higher. Beginning with version 6.0, IPython stopped supporting compatibility with Python versions lower than 3.3 including all versions of Python 2.7.

If you are looking for an IPython version compatible with Python 2.7, please use the IPython 5.x LTS release and refer to its documentation (LTS is the long term support release).

Built-in magic commands

❗ Note

To Jupyter users: Magics are specific to and provided by the IPython kernel. Whether magics are available on a kernel is a decision that is made by the kernel developer on a per-kernel basis. To work properly, Magics must use a syntax element which is not valid in the underlying language. For example, the IPython kernel uses the `%` syntax element for magics as `%` is not a valid unary operator in Python. While, the syntax element has meaning in other languages.

Here is the help auto generated from the docstrings of all the available magics function that IPython ships with.

You can create an register your own magics with IPython. You can find many user defined magics on [PyPI](#). Feel free to publish your own and use the `Framework :: IPython` trove classifier.

Line magics

`%alias`

Define an alias for a system command.

`'%alias alias_name cmd'` defines `'alias_name'` as an alias for `'cmd'`

Then, typing `'alias_name params'` will execute the system command `'cmd params'` (from your underlying operating system).

Aliases have lower precedence than magic functions and Python normal variables, so if `'foo'` is both a Python variable and an alias, the alias can not be executed until `'del foo'` removes the Python variable.

You can use the `%l` specifier in an alias definition to represent the whole line when the alias is

called `For example`:

```
In [2]: alias bracket echo "Input in brackets: <%l>"
In [3]: bracket hello world
Input in brackets: <hello world>
```

You can also define aliases with parameters using `%s` specifiers (one per parameter):

```
In [1]: alias parts echo first %s second %s
In [2]: %parts A B
first A second B
In [3]: %parts A
Incorrect number of arguments: 2 expected.
parts is an alias to: 'echo first %s second %s'
```

Note that `%l` and `%s` are mutually exclusive. You can only use one or the other in your aliases.

Aliases expand Python variables just like system calls using `!` or `!!` do: all expressions prefixed with `'$'` get expanded. For details of the semantic rules, see PEP-215: <http://www.python.org/peps/pep-0215.html>. This is the library used by IPython for variable expansion. If you want to access a true shell variable, an extra `$` is necessary to prevent its expansion by IPython:

```
In [6]: alias show echo
In [7]: PATH='A Python string'
In [8]: show $PATH
A Python string
In [9]: show $$PATH
/usr/local/lf9560/bin:/usr/local/intel/compiler70/ia32/bin:...
```

You can use the alias facility to access all of `$PATH`. See the `%rehashx` function, which automatically creates aliases for the contents of your `$PATH`.

If called with no parameters, `%alias` prints the current alias table for your system. For posix systems, the default aliases are `'cat'`, `'cp'`, `'mv'`, `'rm'`, `'rmdir'`, and `'mkdir'`, and other platform-specific aliases are added. For windows-based systems, the default aliases are `'copy'`, `'ddir'`, `'echo'`, `'ls'`, `'ldir'`, `'mkdir'`, `'ren'`, and `'rmdir'`.

You can see the definition of alias by adding a question mark in the end:

```
In [1]: cat?
Repr: <alias cat for 'cat'>
```

`%alias_magic`

Create an alias for an existing line or cell magic.

Examples

```
In [1]: %alias_magic t timeit
Created `%t` as an alias for `%timeit`.
Created `%%t` as an alias for `%%timeit`.

In [2]: %t -n1 pass
1 loops, best of 3: 954 ns per loop

In [3]: %%t -n1
...: pass
...:
1 loops, best of 3: 954 ns per loop

In [4]: %alias_magic --cell whereami pwd
UsageError: Cell magic function `%%pwd` not found.
In [5]: %alias_magic --line whereami pwd
Created `%whereami` as an alias for `%pwd`.

In [6]: %whereami
Out[6]: u'/home/testuser'

In [7]: %alias_magic h history -p "-l 30" --line
Created `%h` as an alias for `%history -l 30`.
```

positional arguments:

name Name of the magic to be created. target Name of the existing line or cell magic.

optional arguments:

-l, --line	Create a line magic alias.
-c, --cell	Create a cell magic alias.
-p <i>PARAMS</i> , --params <i>PARAMS</i>	
	Parameters passed to the magic function.

%autocall

Make functions callable without having to type parentheses.

Usage:

```
%autocall [mode]
```

The mode can be one of: 0->Off, 1->Smart, 2->Full. If not given, the value is toggled on and off

In more detail, these values mean:

0 -> fully disabled

1 -> active, but do not apply if there are no arguments on the line.

In this mode, you get:

```
In [1]: callable
Out[1]: <built-in function callable>

In [2]: callable 'hello'
-----> callable('hello')
Out[2]: False
```

2 -> Active always. Even if no arguments are present, the callable object is called:

```
In [2]: float
-----> float()
Out[2]: 0.0
```

Note that even with autocall off, you can still use '/' at the start of a line to treat the first argument on the command line as a function and add parentheses to it:

```
In [8]: /str 43
-----> str(43)
Out[8]: '43'
```

all-random (note for auto-testing)

%automagic

Make magic functions callable without having to type the initial %.

Without arguments toggles on/off (when off, you must call it as %automagic, of course). With arguments it sets the value, and you can use any of (case insensitive):

- on, 1, True: to activate
- off, 0, False: to deactivate.

Note that magic functions have lowest priority, so if there's a variable whose name collides with that of a magic fn, automagic won't work for that function (you get the variable instead).

However, if you delete the variable (del var), the previously shadowed magic function becomes visible to automagic again.

%bookmark <name> - set bookmark to current dir
%bookmark <name> <dir> - set bookmark to <dir>
%bookmark -l - list all bookmarks
%bookmark -d <name> - remove bookmark
%bookmark -r - remove all bookmarks

You can later on access a bookmarked folder with:

```
%cd -b <name>
```

or simply '`%cd <name>`' if there is no directory called <name> AND there is such a bookmark defined.

Your bookmarks persist through IPython sessions, but they are associated with each profile.

%cd

Change the current working directory.

This command automatically maintains an internal list of directories you visit during your IPython session, in the variable `_dh`. The command `%dhist` shows this history nicely formatted. You can also do '`cd -<tab>`' to see directory history conveniently.

Usage:

`cd 'dir'`: changes to directory 'dir'.

`cd -`: changes to the last visited directory.

`cd -<n>`: changes to the n-th directory in the directory history.

`cd -foo`: change to directory that matches 'foo' in history

`cd -b <bookmark_name>`: jump to a bookmark set by `%bookmark`

(note: `cd <bookmark_name>` is enough if there is no

directory <bookmark_name>, but a bookmark with the name exists.) '`cd -b <tab>`' allows you to tab-complete bookmark names.

Options:

`-q`: quiet. Do not print the working directory after the `cd` command is executed. By default IPython's `cd` command does print this directory, since the default prompts do not display path information.

Note that `!cd` doesn't work for this purpose because the shell where `!command` runs is immediately discarded after executing 'command'.

Examples

```
In [10]: cd parent/child  
/home/tsuser/parent/child
```

%colors

Switch color scheme for prompts, info system and exception handlers.

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

Examples

To get a plain black and white terminal:

```
%colors nocolor
```

%config

configure IPython

```
%config Class[.trait=value]
```

This magic exposes most of the IPython config system. Any Configurable class should be able to be configured with the simple line:

```
%config Class.trait=value
```

Where **value** will be resolved in the user's namespace, if it is an expression or variable name.

Examples

To see what classes are available for config, pass no arguments:

```
In [1]: %config  
Available objects for config:  
TerminalInteractiveShell  
HistoryManager  
PrefilterManager  
AliasManager  
IPCompleter  
DisplayFormatter
```

To view what is configurable on a given class, just pass the class name:

```
In [2]: %config IPCompleter
```

```
IPCompleter options
```

```
IPCompleter.omit__names=<Enum>
```

```
Current: 2
```

```
Choices: (0, 1, 2)
```

```
Instruct the completer to omit private method names
```

```
Specifically, when completing on ``object.<tab>``.
```

```
When 2 [default]: all names that start with '_' will be excluded.
```

```
When 1: all 'magic' names (``__foo__``) will be excluded.
```

```
When 0: nothing will be excluded.
```

```
IPCompleter.merge_completions=<CBool>
```

```
Current: True
```

```
Whether to merge completion results into a single list
```

```
If False, only the completion results from the first non-empty  
completer will be returned.
```

```
IPCompleter.limit_to__all__=<CBool>
```

```
Current: False
```

```
Instruct the completer to use __all__ for the completion
```

```
Specifically, when completing on ``object.<tab>``.
```

```
When True: only those names in obj.__all__ will be included.
```

```
When False [default]: the __all__ attribute is ignored
```

```
IPCompleter.greedy=<CBool>
```

```
Current: False
```

```
Activate greedy completion
```

```
This will enable completion on elements of lists, results of  
function calls, etc., but can be unsafe because the code is  
actually evaluated on TAB.
```

but the real use is in setting values:

```
In [3]: %config IPCompleter.greedy = True
```

and these values are read from the user_ns if they are variables:

```
In [4]: feeling_greedy=False
```

```
In [5]: %config IPCompleter.greedy = feeling_greedy
```

%debug

```
%debug [--breakpoint FILE:LINE] [statement [statement ...]]
```

Activate the interactive debugger.

This magic command support two ways of activating debugger. One is to activate debugger before executing code. This way, you can set a break point, to step through the code from the point. You can use this mode by giving statements to execute and optionally a breakpoint.

The other one is to activate debugger in post-mortem mode. You can activate this mode simply

If you want IPython to automatically do this on every exception, see the `%pdb` magic for more details.

positional arguments:

statement Code to run in debugger. You can omit this in cell magic mode.

optional arguments:

--breakpoint <FILE:LINE>, -b <FILE:LINE>

Set break point at LINE in FILE.

%dhist

Print your history of visited directories.

`%dhist ->` print full history
`%dhist n ->` print last n entries only
`%dhist n1 n2 ->` print entries between n1 and n2 (n2 not included)

This history is automatically maintained by the `%cd` command, and always available as the global list variable `_dh`. You can use `%cd -<n>` to go to directory number `<n>`.

Note that most of time, you should view directory history by entering `cd -<TAB>`.

%dirs

Return the current directory stack.

%doctest_mode

Toggle doctest mode on and off.

This mode is intended to make IPython behave as much as possible like a plain Python shell, from the perspective of how its prompts, exceptions and output look. This makes it easy to copy and paste parts of a session into doctests. It does so by:

- Changing the prompts to the classic `>>>` ones.
- Changing the exception reporting mode to 'Plain'.
- Disabling pretty-printing of output.

Note that IPython also supports the pasting of code snippets that have leading `'>>>'` and `'<<<'` prompts in them. This means that you can paste doctests from files or docstrings (even if they

have leading whitespace), and the code will execute correctly. You can then use `%history -t` to see the translated history; this will give you the input after removal of all the leading prompts and whitespace, which can be pasted back into an editor.

With these features, you can switch into this mode easily whenever you need to do testing and changes to doctests, without having to leave your existing IPython session.

%edit

Bring up an editor and execute the resulting code.

Usage:

```
%edit [options] [args]
```

`%edit` runs IPython's editor hook. The default version of this hook is set to call the editor specified by your `$EDITOR` environment variable. If this isn't found, it will default to `vi` under Linux/Unix and to `notepad` under Windows. See the end of this docstring for how to change the editor hook.

You can also set the value of this editor via the `TerminalInteractiveShell.editor` option in your configuration file. This is useful if you wish to use a different editor from your typical default with IPython (and for Windows users who typically don't set environment variables).

This command allows you to conveniently edit multi-line code right in your IPython session.

If called without arguments, `%edit` opens up an empty editor with a temporary file and will execute the contents of this file when you close it (don't forget to save it!).

Options:

`-n <number>`: open the editor at a specified line number. By default, the IPython editor hook uses the unix syntax `'editor +N filename'`, but you can configure this by providing your own modified hook if your favorite editor supports line-number specifications with a different syntax.

`-p`: this will call the editor with the same data as the previous time it was used, regardless of how long ago (in your current session) it was.

`-r`: use 'raw' input. This option only applies to input taken from the user's history. By default, the 'processed' history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed as the command line is used instead. When you exit the editor, it will be executed by IPython's own processor.

`-x`: do not execute the edited code immediately upon exit. This is mainly useful if you are editing programs which need to be called with command line arguments, which you can then do using `%run`.

Arguments:

If arguments are given, the following possibilities exist:

- If the argument is a filename, IPython will load that into the editor. It will execute its contents

- The arguments are ranges of input history, e.g. “7 ~1/4-6”. The syntax is the same as in the `%history` magic.
- If the argument is a string variable, its contents are loaded into the editor. You can thus edit any string which contains python code (including the result of previous edits).
- If the argument is the name of an object (other than a string), IPython will try to locate the file where it was defined and open the editor at the point where it is defined. You can use `%edit function` to load an editor exactly at the point where ‘function’ is defined, edit it and have the file be executed automatically.
- If the object is a macro (see `%macro` for details), this opens up your specified editor with a temporary file containing the macro’s data. Upon exit, the macro is reloaded with the contents of the file.

Note: opening at an exact line is only supported under Unix, and some editors (like `kedit` and `gedit` up to Gnome 2.8) do not understand the ‘+NUMBER’ parameter necessary for this feature. Good editors like (X)Emacs, vi, jed, pico and joe all do.

After executing your code, `%edit` will return as output the code you typed in the editor (except when it was an existing file). This way you can reload the code in further invocations of `%edit` as a variable, via `_<NUMBER>` or `Out[<NUMBER>]`, where `<NUMBER>` is the prompt number of the output.

Note that `%edit` is also available through the alias `%ed`.

This is an example of creating a simple function inside the editor and then modifying it. First, start up the editor:

```
In [1]: edit
Editing... done. Executing edited code...
Out[1]: 'def foo():\n    print "foo() was defined in an editing
session"\n'
```

We can then call the function `foo()`:

```
In [2]: foo()
foo() was defined in an editing session
```

Now we edit `foo`. IPython automatically loads the editor with the (temporary) file where `foo()` was previously defined:

```
In [3]: edit foo
Editing... done. Executing edited code...
```

And if we call `foo()` again we get the modified version:

```
In [4]: foo()
foo() has now been changed!
```

Here is an example of how to edit a code snippet successive times. First we call the editor:

```
In [5]: edit
Editing... done. Executing edited code...
hello
Out[5]: "print 'hello'\n"
```

Now we call it again with the previous output (stored in `_`):

```
In [6]: edit _
Editing... done. Executing edited code...
hello world
Out[6]: "print 'hello world'\n"
```

Now we call it with the output #8 (stored in `_8`, also as `Out[8]`):

```
In [7]: edit _8
Editing... done. Executing edited code...
hello again
Out[7]: "print 'hello again'\n"
```

Changing the default editor hook:

If you wish to write your own editor hook, you can put it in a configuration file which you load at startup time. The default hook is defined in the `IPython.core.hooks` module, and you can use that as a starting example for further modifications. That file also has general instructions on how to set a new hook for use once you've defined it.

%env

Get, set, or list environment variables.

Usage:

`%env`: lists all environment variables/values
`%env var`: get value for var
`%env var val`: set value for var
`%env var=val`: set value for var
`%env var=$val`: set value for var, using python expansion if possible

%gui

Enable or disable IPython GUI event loop integration.

This magic replaces IPython's threaded shells that were activated using the (pylab/wthread/etc.) command line flags. GUI toolkits can now be enabled at runtime and keyboard interrupts should work without any problems. The following toolkits are supported: wxPython, PyQt4, PyGTK, Tk and Cocoa (OSX):

```
%gui wx      # enable wxPython event loop integration
%gui qt4|qt   # enable PyQt4 event loop integration
%gui qt5      # enable PyQt5 event loop integration
%gui gtk      # enable PyGTK event loop integration
%gui gtk3     # enable Gtk3 event loop integration
%gui tk       # enable Tk event loop integration
%gui osx      # enable Cocoa event loop integration
              # (requires %matplotlib 1.1)
%gui          # disable all event loop integration
```

WARNING: after any of these has been called you can simply create an application object, but DO NOT start the event loop yourself, as we have already handled that.

%history

```
%history [-n] [-o] [-p] [-t] [-f FILENAME] [-g [PATTERN [PATTERN ...]]]
        [-l [LIMIT]] [-u]
        [range [range ...]]
```

Print input history (`_i<n>` variables), with most recent last.

By default, input history is printed without line numbers so it can be directly pasted into an editor. Use `-n` to show them.

By default, all input history from the current session is displayed. Ranges of history can be indicated using the syntax:

4

Line 4, current session

4-6

Lines 4-6, current session

243/1-5

Lines 1-5, session 243

~2/7

Line 7, session 2 before current

~8/1--~6/5

From the first line of 8 sessions ago, to the fifth line of 6 sessions ago.

Multiple ranges can be entered, separated by spaces

The same syntax is used by %macro, %save, %edit, %rerun

Examples

```
In [6]: %history -n 4-6
4:a = 12
5:print a**2
6:%history -n 4-6
```

positional arguments:

range

optional arguments:

-n	print line numbers for each input. This feature is only available if numbered prompts
-o	also print outputs for each input.
-p	print classic '>>>' python prompts before each input. This is useful for making docun
-t	print the 'translated' history, as IPython understands it. IPython filters your input an
-f <i>FILENAME</i>	<i>FILENAME</i> : instead of printing the output to the screen, redirect it to the given file.
-g <[<i>PATTERN</i> [<i>PATTERN</i> ...]]>	
	treat the arg as a glob pattern to search for in (full) history. This includes the saved h
-l <[<i>LIMIT</i>]>	get the last n lines from all sessions. Specify n as a single arg, or the default is the last
-u	when searching history using <code>-g</code> , show only unique history.

%killbgscripts

Kill all BG processes started by %%script and its family.

%load

Load code into the current frontend.

Usage:

%load [options] source

where source can be a filename, URL, input history range, macro, or element in the user

Options:

- r <lines>: Specify lines or ranges of lines to load from the source. Ranges could be specified as x-y (x..y) or in python-style x:y (x..(y-1)). Both limits x and y can be left blank (meaning the beginning and end of the file, respectively).
- s <symbols>: Specify function or classes to load from python source.
- y : Don't ask confirmation for loading source above 200 000 characters.
- n : Include the user's namespace when searching for source code.

This magic command can either take a local filename, a URL, an history range (see %history) or a macro as argument, it will prompt for confirmation before loading source with more than 200 000 characters, unless -y flag is passed or if the frontend does not support raw_input:

```
%load myscript.py
%load 7-27
%load myMacro
%load http://www.example.com/myscript.py
%load -r 5-10 myscript.py
%load -r 10-20,30,40: foo.py
%load -s MyClass,wonder_function myscript.py
%load -n MyClass
%load -n my_module.wonder_function
```

%load_ext

Load an IPython extension by its module name.

%loadpy

Alias of `%load`

`%loadpy` has gained some flexibility and dropped the requirement of a `.py` extension. So it has been renamed simply into %load. You can look at `%load`'s docstring for more info.

%logoff

Temporarily stop logging.

You must have previously started logging.

%logon

Restart logging.

%logstart

Start logging anywhere in a session.

`%logstart [-o|-r|-t|-q] [log_name [log_mode]]`

If no name is given, it defaults to a file named 'ipython_log.py' in your current directory, in 'rotate' mode (see below).

'%logstart name' saves to file 'name' in 'backup' mode. It saves your history up to that point and then continues logging.

%logstart takes a second optional parameter: logging mode. This can be one of (note that the modes are given unquoted):

append

Keep logging at the end of any existing file.

backup

Rename any existing file to name~ and start name.

global

Append to a single logfile in your home directory.

over

Overwrite any existing log.

rotate

Create rotating logs: name.1~, name.2~, etc.

Options:

Since this marker is always the same, filtering only the output from a log is very easy, using for example a simple awk call:

```
awk -F'#\[Out\]# ' '{if($2) {print $2}}' ipython_log.py
```

-r	log 'raw' input. Normally, IPython's logs contain the processed input, so that user lines are logged
-t	put timestamps before each input line logged (these are put in comments).
-q	suppress output of logstate message when logging is invoked

%logstate

Print the status of the logging system.

%logstop

Fully stop logging and close log file.

In order to start logging again, a new %logstart call needs to be made, possibly (though not necessarily) with a new filename, mode and other options.

%lsmagic

List currently available magic functions.

%macro

Define a macro for future re-execution. It accepts ranges of history, filenames or string objects.

Usage:

%macro [options] name n1-n2 n3-n4 ... n5 .. n6 ...

Options:

-r: use 'raw' input. By default, the 'processed' history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed at the command line is used instead.

-q: quiet macro definition. By default, a tag line is printed to indicate the macro has been created, and then the contents of the macro are printed. If this option is given, then no printout is produced once the macro is created.

This will define a global variable called `name` which is a string made of joining the slices and lines you specify (n1,n2,... numbers above) from your input history into a single string. This variable acts like an automatic function which re-executes those lines as if you had typed them. You just type 'name' at the prompt and the code executes.

The syntax for indicating input ranges is described in %history.

Note: as a 'hidden' feature, you can also use traditional python slice notation, where N:M means numbers N through M-1.

For example, if your history contains (print using `%hist -n`):

```
44: x=1
45: y=3
46: z=x+y
47: print x
48: a=5
49: print 'x',x,'y',y
```

you can create a macro with lines 44 through 47 (included) and line 49 called `my_macro` with:

```
In [55]: %macro my_macro 44-47 49
```

Now, typing `my_macro` (without quotes) will re-execute all this code in one pass.

You don't need to give the line-numbers in order, and any given line number can appear multiple times. You can assemble macros with any lines from your input history in any order.

The macro is a simple object which holds its value in an attribute, but IPython's display system checks for macros and executes them as code instead of printing them when you type their name.

You can view a macro's contents by explicitly printing it with:

```
print macro_name
```

%magic

Print information about the magic function system.

Supported formats: -latex, -brief, -rest

```
%matplotlib
```

```
%matplotlib [-l] [gui]
```

This function lets you activate matplotlib interactive support at any point during an IPython session. It does not import anything into the interactive namespace.

If you are using the inline matplotlib backend in the IPython Notebook you can set which figure formats are enabled using the following:

```
In [1]: from IPython.display import set_matplotlib_formats  
  
In [2]: set_matplotlib_formats('pdf', 'svg')
```

The default for inline figures sets `bbox_inches` to 'tight'. This can cause discrepancies between the displayed image and the identical image created using `savefig`. This behavior can be disabled using the `%config` magic:

```
In [3]: %config InlineBackend.print_figure_kwargs = {'bbox_inches':None}
```

In addition, see the docstring of `IPython.display.set_matplotlib_formats` and `IPython.display.set_matplotlib_close` for more information on changing additional behaviors of the inline backend.

Examples

To enable the inline backend for usage with the IPython Notebook:

```
In [1]: %matplotlib inline
```

In this case, where the matplotlib default is TkAgg:

```
In [2]: %matplotlib  
Using matplotlib backend: TkAgg
```

But you can explicitly request a different GUI backend:

```
In [3]: %matplotlib qt
```

You can list the available backends using the `-l/-list` option:

```
In [4]: %matplotlib --list
```

```
Available matplotlib backends: ['osx', 'qt4', 'qt5', 'gtk3', 'notebook', 'wx', 'qt',
'nbagg',
'gtk', 'tk', 'inline']
```

positional arguments:

gui Name of the matplotlib backend to use ('agg', 'gtk', 'gtk3',

'inline', 'ipympl', 'nbagg', 'notebook', 'osx', 'qt', 'qt4', 'qt5', 'tk', 'widget', 'wx'). If given, the corresponding matplotlib backend is used, otherwise it will be matplotlib's default (which you can set in your matplotlib config file).

optional arguments:

-l, --list	Show available matplotlib backends
------------	------------------------------------

%notebook

```
%notebook filename
```

Export and convert IPython notebooks.

This function can export the current IPython history to a notebook file. For example, to export the history to “foo.ipynb” do “%notebook foo.ipynb”.

The -e or -export flag is deprecated in IPython 5.2, and will be removed in the future.

positional arguments:

filename Notebook name or filename

%page

Pretty print the object and display it through a pager.

%page [options] OBJECT

If no object is given, use _ (last output).

Options:

-r: page str(object), don't pretty-print it.

%pastebin

Upload code to Github's Gist paste bin, returning the URL.

Usage:

`%pastebin [-d "Custom description"] 1-7`

The argument can be an input history range, a filename, or the name of a string or macro.

Options:

-d: Pass a custom description for the gist. The default will say
"Pasted from IPython".

`%pdb`

Control the automatic calling of the pdb interactive debugger.

Call as '`%pdb on`', '`%pdb 1`', '`%pdb off`' or '`%pdb 0`'. If called without argument it works as a toggle.

When an exception is triggered, IPython can optionally call the interactive pdb debugger after the traceback printout. `%pdb` toggles this feature on and off.

The initial state of this feature is set in your configuration file (the option is

`InteractiveShell.pdb`).

If you want to just activate the debugger AFTER an exception has fired, without having to type '`%pdb on`' and rerunning your code, you can use the `%debug` magic.

`%pdef`

Print the call signature for any callable object.

If the object is a class, print the constructor information.

Examples

```
In [3]: %pdef urllib.urlopen
urllib.urlopen(url, data=None, proxies=None)
```

`%pdoc`

Print the docstring for an object.

If the given object is a class, it will print both the class and the constructor docstrings.

`%pfile`

Print (or run through pager) the file where an object is defined.

The file opens at the line where the object definition begins. IPython will honor the environment variable `PAGER` if set, and otherwise will do its best to print the file in a convenient form.

If the given argument is not an object currently defined, IPython will try to interpret it as a filename (automatically adding a .py extension if needed). You can thus use `%pfile` as a syntax highlighting code viewer.

%pinfo

Provide detailed information about an object.

'%pinfo object' is just a synonym for `object?` or `?object`.

%pinfo2

Provide extra detailed information about an object.

'%pinfo2 object' is just a synonym for `object??` or `??object`.

%pip

Intercept usage of `pip` in IPython and direct user to run command outside of IPython.

%popd

Change to directory popped off the top of the stack.

%pprint

Toggle pretty printing on/off.

%precision

Set floating point precision for pretty printing.

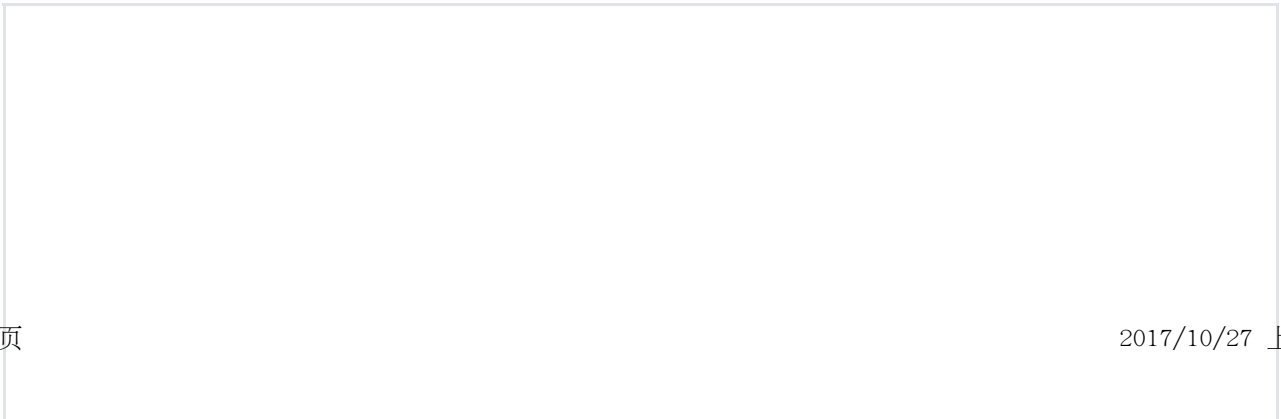
Can set either integer precision or a format string.

If numpy has been imported and precision is an int, numpy display precision will also be set, via

`numpy.set_printoptions`.

If no argument is given, defaults will be restored.

Examples



```
In [1]: from math import pi
```

```
In [2]: %precision 3  
Out[2]: u'%.3f'
```

```
In [3]: pi  
Out[3]: 3.142
```

```
In [4]: %precision %i  
Out[4]: u'%i'
```

```
In [5]: pi  
Out[5]: 3
```

```
In [6]: %precision %e  
Out[6]: u'%e'
```

```
In [7]: pi**10  
Out[7]: 9.364805e+04
```

```
In [8]: %precision  
Out[8]: u'%r'
```

```
In [9]: pi**10  
Out[9]: 93648.047476082982
```

%profile

DEPRECATED since IPython 2.0.

Raise `UsageError`. To profile code use the `%prun` magic.

See Also

`prun` : run code using the Python profiler (`%prun`)

%prun

Run a statement through the python code profiler.

Usage, in line mode:

```
%prun [options] statement
```

Usage, in cell mode:

```
%%prun [options] [statement] code... code...
```

In cell mode, the additional code lines are appended to the (possibly empty) statement in the first line. Cell mode allows you to easily profile multiline blocks without having to put them in a separate function.

The given statement (which doesn't require quote marks) is run via the python profiler in a manner similar to the `profile.run()` function. Namespaces are internally managed to work correctly; `profile.run` cannot be used in IPython because it makes certain assumptions about

Options:

-l <limit>	<p>you can place restrictions on what or how much of the profile gets printed. The</p> <ul style="list-style-type: none">• A string: only information for function names containing this string is printed.• An integer: only these many lines are printed.• A float (between 0 and 1): this fraction of the report is printed (for example, us <p>You can combine several limits with repeated use of the option. For example,</p>																						
-r	return the pstats.Stats object generated by the profiling. This object has all the inform																						
-s <key>	<p>sort profile by given key. You can provide more than one key by using the opti</p> <p>The following is copied verbatim from the profile documentation referenced l</p> <p>When more than one key is provided, additional keys are used as secondary c</p> <p>Abbreviations can be used for any key names, as long as the abbreviation is ur</p> <table><tr><th>Valid Arg</th><th>Meaning</th></tr><tr><td>"calls"</td><td>call count</td></tr><tr><td>"cumulative"</td><td>cumulative time</td></tr><tr><td>"file"</td><td>file name</td></tr><tr><td>"module"</td><td>file name</td></tr><tr><td>"pcalls"</td><td>primitive call count</td></tr><tr><td>"line"</td><td>line number</td></tr><tr><td>"name"</td><td>function name</td></tr><tr><td>"nfl"</td><td>name/file/line</td></tr><tr><td>"stdname"</td><td>standard name</td></tr><tr><td>"time"</td><td>internal time</td></tr></table> <p>Note that all sorts on statistics are in descending order (placing most time cor</p>	Valid Arg	Meaning	"calls"	call count	"cumulative"	cumulative time	"file"	file name	"module"	file name	"pcalls"	primitive call count	"line"	line number	"name"	function name	"nfl"	name/file/line	"stdname"	standard name	"time"	internal time
Valid Arg	Meaning																						
"calls"	call count																						
"cumulative"	cumulative time																						
"file"	file name																						
"module"	file name																						
"pcalls"	primitive call count																						
"line"	line number																						
"name"	function name																						
"nfl"	name/file/line																						
"stdname"	standard name																						
"time"	internal time																						
-T <filename>	save profile results as shown on screen to a text file. The profile is still shown on screen																						
-D <filename>	save (via dump_stats) profile statistics to given filename. This data is in a format under																						
-q	suppress output to the pager. Best used with -T and/or -D above.																						

You can read the complete documentation for the profile module with:

```
In [1]: import profile; profile.help()
```

%psearch

Search for object in namespaces by wildcard.

%psearch [options] PATTERN [OBJECT TYPE]

Note: ? can be used as a synonym for %psearch, at the beginning or at the end: both a*? and ?a* are equivalent to '%psearch a*'. Still, the rest of the command line must be unchanged (options come first), so for example the following forms are equivalent

%psearch -i a* function -i a* function? ?-i a* function

Arguments:

PATTERN

where PATTERN is a string containing * as a wildcard similar to its use in a shell. The pattern is matched in all namespaces on the search path. By default objects starting with a single _ are not matched, many IPython generated objects have a single underscore. The default is case insensitive matching. Matching is also done on the attributes of objects and not only on the objects in a module.

[OBJECT TYPE]

Is the name of a python type from the types module. The name is given in lowercase without the ending type, ex. StringType is written string. By adding a type here only objects matching the given type are matched. Using all here makes the pattern match all types (this is the default).

Options:

-i/-c: make the pattern case insensitive/sensitive. If neither of these options are given, the default is read from your configuration file, with the option

`InteractiveShell.wildcards_case_sensitive`. If this option is not specified in your configuration file, IPython's internal default is to do a case sensitive search.

-e/-s NAMESPACE: exclude/search a given namespace. The pattern you specify can be searched in any of the following namespaces: 'builtin', 'user', 'user_global', 'internal', 'alias', where 'builtin' and 'user' are the search defaults. Note that you should not use quotes when specifying namespaces.

'Builtin' contains the python module builtin, 'user' contains all user data, 'alias' only contain the shell aliases and no python objects, 'internal' contains objects used by IPython. The 'user_global' namespace is only used by embedded IPython instances, and it contains module-level globals. You can add namespaces to the search with -s or exclude them with -e (these options can be given more than once).

Examples

```
%psearch a*          -> objects beginning with an a
%psearch -e builtin a* -> objects NOT in the builtin space starting in a
%psearch a* function -> all functions beginning with an a
%psearch re.e*        -> objects beginning with an e in module re
%psearch r*.e*        -> objects that start with e in modules starting in r
%psearch r*.* string  -> all strings in modules beginning with r
```

Case sensitive search:

```
%psearch -c a*      list all object beginning with lower case a
```

Show objects beginning with a single _:

```
%psearch -a _*      list objects beginning with a single underscore
```

%psource

Print (or run through pager) the source code for an object.

%pushd

Place the current dir on stack and change directory.

Usage:

%pushd['dirname']

Return the current working directory path.

Examples

```
In [9]: pwd
Out[9]: '/home/tsuser/sprint/ipython'
```

%pycat

Show a syntax-highlighted file through a pager.

This magic is similar to the cat utility, but it will assume the file to be Python source and will show it with syntax highlighting.

This magic command can either take a local filename, an url, an history range (see %history) or a macro as argument

```
%pycat myscript.py
%pycat 7-27
%pycat myMacro
%pycat http://www.example.com/myscript.py
```

%pylab

```
%pylab [--no-import-all] [gui]
```

Load numpy and matplotlib to work interactively.

This function lets you activate pylab (matplotlib, numpy and interactive support) at any point during an IPython session.

%pylab makes the following imports:

```
import numpy
import matplotlib
from matplotlib import pylab, mlab, pyplot
np = numpy
plt = pyplot

from IPython.display import display
from IPython.core.pylabtools import figsize, getfigs

from pylab import *
from numpy import *
```

See the `%matplotlib` magic for more details about activating matplotlib without affecting the interactive namespace.

positional arguments:

gui Name of the matplotlib backend to use ('agg', 'gtk',

'gtk3', 'inline', 'ipympl', 'nbagg', 'notebook', 'osx', 'qt', 'qt4', 'qt5', 'tk', 'widget', 'wx'). If given, the corresponding matplotlib backend is used, otherwise it will be matplotlib's default (which you can set in your matplotlib config file).

optional arguments:

`--no-import-all`

Prevent IPython from performing `import *` into the interactive namespace. You can govern the

`%quickref`

Show a quick reference sheet

`%recall`

Repeat a command, or get command to input line for editing.

`%recall` and `%rep` are equivalent.

- `%recall` (no arguments):

Place a string version of last computation result (stored in the special `'_'` variable) to the next input prompt. Allows you to create elaborate command lines without using copy-paste:

```
In[1]: l = ["hei", "vaan"]
In[2]: "".join(l)
Out[2]: heivaan
In[3]: %recall
In[4]: heivaan_ <== cursor blinking
```

`%recall 45`

Place history line 45 on the next input prompt. Use `%hist` to find out the number.

`%recall 1-4`

Combine the specified lines into one cell, and place it on the next input prompt. See `%history` for the slice syntax.

`%recall foo+bar`

If foo+bar can be evaluated in the user namespace, the result is placed at the next input prompt. Otherwise, the history is searched for lines which contain that substring, and the most recent one is placed at the next input prompt.

%rehashx

Update the alias table with all executable files in \$PATH.

rehashx explicitly checks that every entry in \$PATH is a file with execute access (os.X_OK).

Under Windows, it checks executability as a match against a '|'-separated string of extensions, stored in the IPython config variable win_exec_ext. This defaults to 'exe|com|bat'.

This function also resets the root module cache of module completer, used on slow filesystems.

%reload_ext

Reload an IPython extension by its module name.

%rerun

Re-run previous input

By default, you can specify ranges of input history to be repeated (as with %history). With no arguments, it will repeat the last line.

Options:

-l <n> : Repeat the last n lines of input, not including the current command.

-g foo : Repeat the most recent line which contains foo

%reset

Resets the namespace by removing all names defined by the user, if called without arguments, or by removing some types of objects, such as everything currently in IPython's In[] and Out[] containers (see the parameters for details).

Parameters

-f : force reset without asking for confirmation.

-s : 'Soft' reset: Only clears your namespace, leaving history intact.

References to objects may be kept. By default (without this option), we do a 'hard' reset, giving you a new session and removing all references to objects from the current session.

in : reset input history

out : reset output history

dhist : reset directory history

See Also

`reset_selective` : invoked as `%reset_selective`

Examples

```
In [6]: a = 1

In [7]: a
Out[7]: 1

In [8]: 'a' in _ip.user_ns
Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns
Out[1]: False

In [2]: %reset -f in
Flushing input history

In [3]: %reset -f dhist in
Flushing directory history
Flushing input history
```

Notes

Calling this magic from clients that do not implement standard input, such as the ipython notebook interface, will reset the namespace without confirmation.

`%reset_selective`

Resets the namespace by removing names defined by the user.

Input/Output history are left around in case you need them.

`%reset_selective [-f] regex`

No action is taken if regex is not included

Options

`-f` : force reset without asking for confirmation.

See Also

`reset` : invoked as `%reset`

Examples

We first fully reset the namespace so your output looks identical to this example for pedagogical reasons; in practice you do not need a full reset:

```
In [1]: %reset -f
```

Now, with a clean namespace we can make a few variables and use `%reset_selective` to only delete names that match our regexp:

```
In [2]: a=1; b=2; c=3; b1m=4; b2m=5; b3m=6; b4m=7; b2s=8

In [3]: who_ls
Out[3]: ['a', 'b', 'b1m', 'b2m', 'b2s', 'b3m', 'b4m', 'c']

In [4]: %reset_selective -f b[2-3]m

In [5]: who_ls
Out[5]: ['a', 'b', 'b1m', 'b2s', 'b4m', 'c']

In [6]: %reset_selective -f d

In [7]: who_ls
Out[7]: ['a', 'b', 'b1m', 'b2s', 'b4m', 'c']

In [8]: %reset_selective -f c

In [9]: who_ls
Out[9]: ['a', 'b', 'b1m', 'b2s', 'b4m']

In [10]: %reset_selective -f b

In [11]: who_ls
Out[11]: ['a']
```

Notes

Calling this magic from clients that do not implement standard input, such as the ipython notebook interface, will reset the namespace without confirmation.

%run

Run the named file inside IPython as a program.

Usage:

```
%run [-n -i -e -G]
      [(-t [-N<N>] | -d [-b<N>] | -p [profile options] )]
      (-m mod | file ) [args]
```

Parameters after the filename are passed as command-line arguments to the program (put in `sys.argv`). Then, control returns to IPython's prompt.

This is similar to running at a system prompt `python file args`, but with the advantage of giving you IPython's tracebacks, and of loading all variables into your interactive namespace for further

The file is executed in a namespace initially consisting only of `__name__ == '__main__'` and `sys.argv` constructed as indicated. It thus sees its environment as if it were being run as a stand-alone program (except for sharing global objects such as previously imported modules). But after execution, the IPython interactive namespace gets updated with all variables defined in the program (except for `__name__` and `sys.argv`). This allows for very convenient loading of code for interactive work, while giving each program a 'clean sheet' to run in.

Arguments are expanded using shell-like glob match. Patterns `'*'`, `'?'`, `'[seq]'` and `'[!seq]'` can be used. Additionally, tilde `'~'` will be expanded into user's home directory. Unlike real shells, quotation does not suppress expansions. Use *two* back slashes (e.g. `*`) to suppress expansions. To completely disable these expansions, you can use `-G` flag.

Options:

-n	<code>__name__</code> is NOT set to <code>'__main__'</code> , but to the running file's name without extension (as python does)
-i	run the file in IPython's namespace instead of an empty one. This is useful if you are experimenting
-e	ignore <code>sys.exit()</code> calls or <code>SystemExit</code> exceptions in the script being run. This is particularly useful if ll
-t	print timing information at the end of the run. IPython will give you an estimated CPU time consum

If `-t` is given, an additional `-N<N>` option can be given, where `<N>` must be an integer indicating how many times you want the script to run. The final timing report will include total and per run results.

For example (testing the script `uniq_stable.py`):

```
In [1]: run -t uniq_stable

IPython CPU timings (estimated):
  User :    0.19597 s.
  System:    0.0 s.

In [2]: run -t -N5 uniq_stable

IPython CPU timings (estimated):
Total runs performed: 5
  Times :      Total      Per run
  User :    0.910862 s,    0.1821724 s.
  System:    0.0 s,      0.0 s.
```

	<p>run your program under the control of pdb, the Python debugger. This allows you to execute</p> <pre>pdb.run('execfile("YOURFILENAME")')</pre>
-d	<p>with a breakpoint set on line 1 of your file. You can change the line number for this automatic</p> <pre>%run -d -b40 myscript</pre> <p>will set the first breakpoint at line 40 in myscript.py. Note that the first breakpoint must be 1. Or you can specify a breakpoint in a different file:</p> <pre>%run -d -b myotherfile.py:20 myscript</pre> <p>When the pdb debugger starts, you will see a (Pdb) prompt. You must first enter 'c' (with no argument). Entering 'help' gives information about the use of the debugger. You can easily see pdb's documentation.</p>
-p	<p>run program under the control of the Python profiler module (which prints a detailed report). You can pass other options after -p which affect the behavior of the profiler itself. See the documentation for the Python profiler module. In this mode, the program's variables do NOT propagate back to the IPython interactive namespace. Internally this triggers a call to %prun, see its documentation for details on the options and</p>

There is one special usage for which the text above doesn't apply: if the filename ends with .ipy[nb], the file is run as ipython script, just as if the commands were written on IPython prompt.

-m	<p>specify module name to load instead of script path. Similar to the -m option for the python command.</p> <pre>%run -m example</pre> <p>will run the example module.</p>
-G	<p>disable shell-like glob expansion of arguments.</p>

%save

Save a set of lines or a macro to a given filename.

Usage:

%save [options] filename n1-n2 n3-n4 ... n5 .. n6 ...

Options:

-r: use ‘raw’ input. By default, the ‘processed’ history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed as the command line is used instead.

-f: force overwrite. If file exists, %save will prompt for overwrite unless -f is given.

-a: append to the file instead of overwriting it.

This function uses the same syntax as %history for input ranges, then saves the lines to the filename you specify.

It adds a ‘.py’ extension to the file if you don’t do so yourself, and it asks for confirmation before overwriting existing files.

If `-r` option is used, the default extension is `.ipy`.

%sc

Shell capture - run shell command and capture output (DEPRECATED use !).

DEPRECATED. Suboptimal, retained for backwards compatibility.

You should use the form ‘var = !command’ instead. Example:

“%sc -l myfiles = ls ~” should now be written as

“myfiles = !ls ~”

myfiles.s, myfiles.l and myfiles.n still apply as documented below.

%sc [options] varname=command

IPython will run the given command using `commands.getoutput()`, and will then update the user’s interactive namespace with a variable called varname, containing the value of the call. Your command can contain shell wildcards, pipes, etc.

The ‘=’ sign in the syntax is mandatory, and the variable name you supply must follow Python’s standard conventions for valid names.

(A special format without variable name exists for internal use)

Options:

-l: list output. Split the output on newlines into a list before assigning it to the given variable. By default the output is stored as a single string.

-v: verbose. Print the contents of the variable.

In most cases you should not need to split as a list, because the returned value is a special type of string which can automatically provide its contents either as a list (split on newlines) or as a

For example:

```
# Capture into variable a
In [1]: sc a=ls *py

# a is a string with embedded newlines
In [2]: a
Out[2]: 'setup.py\nwin32_manual_post_install.py'

# which can be seen as a list:
In [3]: a.l
Out[3]: ['setup.py', 'win32_manual_post_install.py']

# or as a whitespace-separated string:
In [4]: a.s
Out[4]: 'setup.py win32_manual_post_install.py'

# a.s is useful to pass as a single command line:
In [5]: !wc -l $a.s
146 setup.py
130 win32_manual_post_install.py
276 total

# while the list form is useful to loop over:
In [6]: for f in a.l:
...:     !wc -l $f
...:
146 setup.py
130 win32_manual_post_install.py
```

Similarly, the lists returned by the `-l` option are also special, in the sense that you can equally invoke the `.s` attribute on them to automatically get a whitespace-separated string from their contents:

```
In [7]: sc -l b=ls *py

In [8]: b
Out[8]: ['setup.py', 'win32_manual_post_install.py']

In [9]: b.s
Out[9]: 'setup.py win32_manual_post_install.py'
```

In summary, both the lists and strings used for output capture have the following special attributes:

```
.l (or .list) : value as list.
.n (or .nlstr): value as newline-separated string.
.s (or .spstr): value as space-separated string.
```

%set_env

Set environment variables. Assumptions are that either “val” is a name in the user namespace, or val is something that evaluates to a string.

Usage:

%set_env var val: set value for var
%set_env var=val: set value for var
%set_env var=\$val: set value for var, using python expansion if possible

%sx

Shell execute - run shell command and capture output (!! is short-hand).

%sx command

IPython will run the given command using `commands.getoutput()`, and return the result formatted as a list (split on ‘n’). Since the output is `_returned_`, it will be stored in ipython’s regular output cache `Out[N]` and in the ‘_N’ automatic variables.

Notes:

1) If an input line begins with ‘!!’, then %sx is automatically invoked. That is, while:

```
!!ls
```

causes ipython to simply issue `system('ls')`, typing:

```
!!ls
```

is a shorthand equivalent to:

```
%sx ls
```

2) %sx differs from %sc in that %sx automatically splits into a list, like ‘%sc -l’. The reason for this is to make it as easy as possible to process line-oriented shell output via further python commands. %sc is meant to provide much finer control, but requires more typing.

3) Just like %sc -l, this is a list with special attributes:

```
.l (or .list) : value as list.  
.n (or .nlstr): value as newline-separated string.  
.s (or .spstr): value as whitespace-separated string.
```

This is very useful when trying to use such lists as arguments to system commands.

%system

Shell execute - run shell command and capture output (!! is short-hand).

%sx command

IPython will run the given command using `commands.getoutput()`, and return the result formatted as a list (split on 'n'). Since the output is `_returned_`, it will be stored in ipython's regular output cache `Out[N]` and in the `'_N'` automatic variables.

Notes:

1) If an input line begins with '!!', then %sx is automatically invoked. That is, while:

```
!!ls
```

causes ipython to simply issue `system('ls')`, typing:

```
!!ls
```

is a shorthand equivalent to:

```
%sx ls
```

2) %sx differs from %sc in that %sx automatically splits into a list, like '%sc -l'. The reason for this is to make it as easy as possible to process line-oriented shell output via further python commands. %sc is meant to provide much finer control, but requires more typing.

3) Just like %sc -l, this is a list with special attributes:

```
.l (or .list) : value as list.  
.n (or .nlstr): value as newline-separated string.  
.s (or .spstr): value as whitespace-separated string.
```

This is very useful when trying to use such lists as arguments to system commands.

%tb

Print the last traceback with the currently active exception mode.

See %xmode for changing exception reporting modes.

Time execution of a Python statement or expression.

Note that under Win32, system time is always reported as 0, since it can not be measured.

This function can be used both as a line and cell magic:

- In line mode you can time a single-line statement (though multiple ones can be chained with using semicolons).
- In cell mode, you can time the cell body (a directly following statement raises an error).

This function provides very basic timing functionality. Use the `timeit` magic for more control over the measurement.

Examples

```
In [1]: %time 2**128
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00
Out[1]: 340282366920938463463374607431768211456L
```

```
In [2]: n = 1000000
```

```
In [3]: %time sum(range(n))
CPU times: user 1.20 s, sys: 0.05 s, total: 1.25 s
Wall time: 1.37
Out[3]: 499999500000L
```

```
In [4]: %time print 'hello world'
hello world
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00
```

Note that the time needed by Python to `compile` the given expression will be reported **if** it **is** more than `0.1s`. In this example, the actual exponentiation **is** done by Python at compilation time, so **while** the expression can take a noticeable amount of time to compute, that time **is** purely due to the compilation:

```
In [5]: %time 3**9999;
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
```

```
In [6]: %time 3**999999;
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
Compiler : 0.78 s
```

%timeit

Time execution of a Python statement or expression

Usage, in line mode:

```
%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] statement
```

or in cell mode:

```
%%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] setup_code code code...
```

Time execution of a Python statement or expression using the timeit module. This function can be used both as a line and cell magic:

- In line mode you can time a single-line statement (though multiple ones can be chained with using semicolons).
- In cell mode, the statement in the first line is used as setup code (executed but not timed) and the body of the cell is timed. The cell body has access to any variables created in the setup code.

Options: -n<N>: execute the given statement <N> times in a loop. If this value is not given, a fitting value is chosen.

-r<R>: repeat the loop iteration <R> times and take the best result. Default: 3

-t: use time.time to measure the time, which is the default on Unix. This function measures wall time.

-c: use time.clock to measure the time, which is the default on Windows and measures wall time. On Unix, resource.getrusage is used instead and returns the CPU user time.

-p<P>: use a precision of <P> digits to display the timing result. Default: 3

-q: Quiet, do not print result.

-o: return a TimeitResult that can be stored in a variable to inspect the result in more details.

Examples

```
In [1]: %timeit pass
8.26 ns ± 0.12 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)

In [2]: u = None

In [3]: %timeit u is None
29.9 ns ± 0.643 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [4]: %timeit -r 4 u == None

In [5]: import time

In [6]: %timeit -n1 time.sleep(2)
```

The times reported by %timeit will be slightly higher than those reported by the timeit.py script when variables are accessed. This is due to the fact that %timeit executes the statement in the namespace of the shell, compared with timeit.py, which uses a single setup statement to import function or create variables. Generally, the bias does not matter as long as results from timeit.py are not mixed with those from %timeit.

%unload_ext

Unload an IPython extension by its module name.

Not all extensions can be unloaded, only those which define an `unload_ipython_extension` function.

%who

Print all interactive variables, with some minimal formatting.

If any arguments are given, only variables whose type matches one of these are printed. For example:

```
%who function str
```

will only list functions and strings, excluding all other types of variables. To find the proper type names, simply use `type(var)` at a command line to see how python prints type names. For example:

```
In [1]: type('hello')\  
Out[1]: <type 'str'>
```

indicates that the type name for strings is 'str'.

`%who` always excludes executed names loaded through your configuration file and things which are internal to IPython.

This is deliberate, as typically you may load many modules and the purpose of `%who` is to show you only what you've manually defined.

Examples

Define two variables and list them with who:

```
In [1]: alpha = 123  
  
In [2]: beta = 'test'  
  
In [3]: %who  
alpha    beta  
  
In [4]: %who int  
alpha  
  
In [5]: %who str  
beta
```

%who_ls

Return a sorted list of all interactive variables.

If arguments are given, only variables of types matching these arguments are returned.

Examples

Define two variables and list them with who_ls:

```
In [1]: alpha = 123

In [2]: beta = 'test'

In [3]: %who_ls
Out[3]: ['alpha', 'beta']

In [4]: %who_ls int
Out[4]: ['alpha']

In [5]: %who_ls str
Out[5]: ['beta']
```

%whos

Like %who, but gives some extra information about each variable.

The same type filtering of %who can be applied here.

For all variables, the type is printed. Additionally it prints:

- For {},[],(): their length.
- For numpy arrays, a summary with shape, number of elements, typecode and size in memory.
- Everything else: a string representation, snipping their middle if too long.

Examples

Define two variables and list them with whos:

```
In [1]: alpha = 123

In [2]: beta = 'test'

In [3]: %whos
Variable  Type      Data/Info
alpha     int       123
beta      str       test
```


%xdel

Delete a variable, trying to clear it from anywhere that IPython's machinery has references to it. By default, this uses the identity of the named object in the user namespace to remove references held under other names. The object is also removed from the output history.

Options

-n : Delete the specified name from all namespaces, without checking their identity.

%xmode

Switch modes for the exception handlers.

Valid modes: Plain, Context and Verbose.

If called without arguments, acts as a toggle.

Cell magics

%%bash

%%bash script magic

Run cells with bash in a subprocess.

This is a shortcut for `%%script bash`

%%capture

```
%capture [--no-stderr] [--no-stdout] [--no-display] [output]
```

run the cell, capturing stdout, stderr, and IPython's rich display() calls.

positional arguments:

output The name of the variable in which to store output. This is a

utils.io.CapturedIO object with stdout/err attributes for the text of the captured output. CapturedOutput also has a show() method for displaying the output, and __call__ as well, so you can use that to quickly display the output. If unspecified, captured output is discarded.

optional arguments:

--no-stderr	Don't capture stderr.
--no-stdout	Don't capture stdout.
--no-display	Don't capture IPython's rich display.

%html

Render the cell as a block of HTML

%%javascript

Run the cell block of Javascript code

%%js

Run the cell block of Javascript code

Alias of `%%javascript`

%%latex

Render the cell as a block of latex

The subset of latex which is support depends on the implementation in the client. In the Jupyter Notebook, this magic only renders the subset of latex defined by MathJax [here] (<https://docs.mathjax.org/en/v2.5-latest/tex.html>).

%%markdown

Render the cell as Markdown text block

%%perl

%%perl script magic

Run cells with perl in a subprocess.

This is a shortcut for `%%script perl`

%%pypy

%%pypy script magic

Run cells with pypy in a subprocess.

This is a shortcut for `%%script pypy`

%%python

%%python script magic

Run cells with python in a subprocess.

This is a shortcut for `%%script python`

%%python2

%%python2 script magic

Run cells with python2 in a subprocess.

This is a shortcut for `%%script python2`

%%python3

%%python3 script magic

Run cells with python3 in a subprocess.

This is a shortcut for `%%script python3`

%%ruby

%%ruby script magic

Run cells with ruby in a subprocess.

This is a shortcut for `%%script ruby`

%%script

```
%shebang [--proc PROC] [--bg] [--err ERR] [--out OUT]
```

Run a cell via a shell command

The `%%script` line is like the `#!` line of script, specifying a program (bash, perl, ruby, etc.) with which to run.

The rest of the cell is run by that program.

Examples

```
In [1]: %%script bash
...: for i in 1 2 3; do
...:   echo $i
...: done
1
2
3
```

optional arguments:

<code>--proc PROC</code>	The variable in which to store Popen instance. This is used only when <code>-bg</code> option is g
--------------------------	--

--bg Whether to run the script in the background. If given, the only way to see the output

--err *ERR* The variable in which to store stderr from the script. If the script is backgrounded, th

--out *OUT* The variable in which to store stdout from the script. If the script is backgrounded, tl

%%sh

%%sh script magic

Run cells with sh in a subprocess.

This is a shortcut for `%%script sh`

%%svg

Render the cell as an SVG literal

%%writefile

```
%%writefile [-a] filename
```

Write the contents of the cell to a file.

The file will be overwritten unless the -a (–append) flag is specified.

positional arguments:

filename file to write

optional arguments:

-a, --append	Append contents of the cell to an existing file. The file will be created if it does not ex
--------------	---