
IPython Documentation

Release 0.11.dev

The IPython Development Team

February 10, 2011

CONTENTS

1	Introduction	1
1.1	Overview	1
1.2	Enhanced interactive Python shell	1
1.3	Interactive parallel computing	3
2	What's new in IPython	5
2.1	Development version	5
2.2	0.10 series	8
2.3	0.9 series	14
2.4	0.8 series	18
3	Installation	21
3.1	Overview	21
3.2	Quickstart	21
3.3	Installing IPython itself	22
3.4	Basic optional dependencies	23
3.5	Dependencies for IPython.kernel (parallel computing)	25
3.6	Dependencies for IPython.frontend (the IPython GUI)	26
4	Using IPython for interactive work	27
4.1	Quick IPython tutorial	27
4.2	IPython reference	33
4.3	IPython as a system shell	57
4.4	IPython as a QtGUI widget	62
5	Using IPython for parallel computing	69
5.1	Overview and getting started	69
5.2	Starting the IPython controller and engines	73
5.3	IPython's multiengine interface	79
5.4	The IPython task interface	93
5.5	Using MPI with IPython	95
5.6	Security details of IPython	98
5.7	Getting started with Windows HPC Server 2008	104
5.8	Parallel examples	112

6	Configuration and customization	123
6.1	Overview of the IPython configuration system	123
6.2	IPython extensions	128
6.3	IPython plugins	129
6.4	Configuring the ipython command line application	130
6.5	Editor configuration	132
6.6	Outdated configuration information that might still be useful	134
7	IPython developer's guide	139
7.1	How to contribute to IPython	139
7.2	Working with <i>ipython</i> source code	140
7.3	Coding guide	152
7.4	Documenting IPython	154
7.5	Testing IPython for users and developers	156
7.6	Releasing IPython	163
7.7	Development roadmap	163
7.8	IPython module organization	165
7.9	Messaging in IPython	166
7.10	Messaging for Parallel Computing	182
7.11	Connection Diagrams of The IPython ZMQ Cluster	187
7.12	The magic commands subsystem	192
7.13	IPython.kernel.core.notification blueprint	196
7.14	Notes on code execution in <i>InteractiveShell</i>	198
7.15	IPython Qt interface	199
7.16	Porting IPython to a two process model using zeromq	201
8	The IPython API	203
8.1	config.loader	203
8.2	core.alias	209
8.3	core.application	212
8.4	core.autocall	217
8.5	core.builtin_trap	218
8.6	core.compilerop	219
8.7	core.completer	220
8.8	core.completerlib	226
8.9	core.crashhandler	227
8.10	core.debugger	229
8.11	core.display	237
8.12	core.display_trap	239
8.13	core.displayhook	240
8.14	core.displaypub	243
8.15	core.error	246
8.16	core.excolors	248
8.17	core.extensions	248
8.18	core.formatters	250
8.19	core.history	271
8.20	core.hooks	275
8.21	core.inputsplitter	277

8.22	core.interactiveshell	284
8.23	core.ipapi	324
8.24	core.logger	324
8.25	core.macro	325
8.26	core.magic	326
8.27	core.oinspect	348
8.28	core.page	353
8.29	core.payload	355
8.30	core.payloadpage	356
8.31	core.plugin	357
8.32	core.prefilter	360
8.33	core.prompts	396
8.34	core.splitinput	399
8.35	core.ultratb	399
8.36	kernel.clientconnector	411
8.37	kernel.clientinterfaces	420
8.38	kernel.codeutil	425
8.39	kernel.controllerservice	425
8.40	kernel.core.display_formatter	432
8.41	kernel.core.display_trap	433
8.42	kernel.core.error	434
8.43	kernel.core.fd_redirector	436
8.44	kernel.core.file_like	437
8.45	kernel.core.history	438
8.46	kernel.core.interpreter	439
8.47	kernel.core.macro	444
8.48	kernel.core.magic	445
8.49	kernel.core.message_cache	446
8.50	kernel.core.output_trap	447
8.51	kernel.core.prompts	448
8.52	kernel.core.redirector_output_trap	452
8.53	kernel.core.sync_traceback_trap	453
8.54	kernel.core.traceback_formatter	454
8.55	kernel.core.traceback_trap	455
8.56	kernel.core.util	456
8.57	kernel.engineconnector	459
8.58	kernel.enginefc	460
8.59	kernel.engineservice	468
8.60	kernel.error	489
8.61	kernel.ipclusterapp	497
8.62	kernel.ipengineapp	501
8.63	kernel.map	505
8.64	kernel.mapper	506
8.65	kernel.multiengine	515
8.66	kernel.multiengineclient	543
8.67	kernel.multienginefc	556
8.68	kernel.newserialized	562
8.69	kernel.parallelfunction	569

8.70	kernel.pbutil	577
8.71	kernel.pendingdeferred	577
8.72	kernel.pickleutil	579
8.73	kernel.twistedutil	580
8.74	kernel.util	584
8.75	lib.backgroundjobs	584
8.76	lib.clipboard	589
8.77	lib.deepreload	590
8.78	lib.demo	590
8.79	lib.guisupport	604
8.80	lib.inpathook	605
8.81	lib.inpathookgtk	609
8.82	lib.inpathookwx	609
8.83	lib.irunner	612
8.84	lib.latextools	618
8.85	lib.pylabtools	619
8.86	testing	621
8.87	testing.decorators	621
8.88	testing.decorators_trial	623
8.89	testing.globalipapp	624
8.90	testing.iptest	626
8.91	testing.ipunittest	628
8.92	testing.mkdoctests	629
8.93	testing.nosepatch	631
8.94	testing.parametric	632
8.95	testing.plugin.dtexample	632
8.96	testing.plugin.show_refs	634
8.97	testing.plugin.simple	635
8.98	testing.plugin.test_ipdoctest	635
8.99	testing.plugin.test_refs	636
8.100	testing.tools	637
8.101	utils.PyColorize	640
8.102	utils.attic	641
8.103	utils.autoattr	643
8.104	utils.coloransi	645
8.105	utils.data	648
8.106	utils.decorators	649
8.107	utils.dir2	649
8.108	utils.doctestreload	650
8.109	utils.frame	651
8.110	utils.generics	651
8.111	utils.growl	652
8.112	utils.importstring	653
8.113	utils.io	653
8.114	utils.ipstruct	656
8.115	utils.jsonutil	660
8.116	utils.notification	661
8.117	utils.path	663

8.118	utils.pickleshare	666
8.119	utils.process	668
8.120	utils.strdispatch	670
8.121	utils.sysinfo	671
8.122	utils.syspathcontext	672
8.123	utils.terminal	673
8.124	utils.text	673
8.125	utils.timing	682
8.126	utils.upgradedir	683
8.127	utils.warn	683
8.128	utils.wildcard	684
9	Frequently asked questions	687
9.1	General questions	687
9.2	Questions about parallel computing with IPython	687
10	About IPython	689
10.1	Credits	689
10.2	History	693
10.3	License and Copyright	694
	Bibliography	697
	Python Module Index	699
	Index	701

INTRODUCTION

1.1 Overview

One of Python's most useful features is its interactive interpreter. This system allows very fast testing of ideas without the overhead of creating test files as is typical in most programming languages. However, the interpreter supplied with the standard Python distribution is somewhat limited for extended interactive use.

The goal of IPython is to create a comprehensive environment for interactive and exploratory computing. To support this goal, IPython has two main components:

- An enhanced interactive Python shell.
- An architecture for interactive parallel computing.

All of IPython is open source (released under the revised BSD license).

1.2 Enhanced interactive Python shell

IPython's interactive shell (**ipython**), has the following goals, amongst others:

1. Provide an interactive shell superior to Python's default. IPython has many features for object introspection, system shell access, and its own special command system for adding functionality when working interactively. It tries to be a very efficient environment both for Python code development and for exploration of problems using Python objects (in situations like data analysis).
2. Serve as an embeddable, ready to use interpreter for your own programs. IPython can be started with a single call from inside another program, providing access to the current namespace. This can be very useful both for debugging purposes and for situations where a blend of batch-processing and interactive exploration are needed. New in the 0.9 version of IPython is a reusable wxPython based IPython widget.
3. Offer a flexible framework which can be used as the base environment for other systems with Python as the underlying language. Specifically scientific environments like Mathematica, IDL and Matlab inspired its design, but similar ideas can be useful in many fields.
4. Allow interactive testing of threaded graphical toolkits. IPython has support for interactive, non-blocking control of GTK, Qt and WX applications via special threading flags. The normal Python shell can only do this for Tkinter applications.

1.2.1 Main features of the interactive shell

- Dynamic object introspection. One can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke ('?', and using '??' provides additional detail).
- Searching through modules and namespaces with '*' wildcards, both when using the '?' system and via the '%psearch' command.
- Completion in the local namespace, by typing TAB at the prompt. This works for keywords, modules, methods, variables and files in the current directory. This is supported via the readline library, and full access to configuring readline's behavior is provided. Custom completers can be implemented easily for different purposes (system commands, magic arguments etc.)
- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.
- User-extensible 'magic' commands. A set of commands prefixed with '%' is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with '!' are passed directly to the system shell, and using '!!' or 'var = !cmd' captures shell output into python variables for further use.
- Background execution of Python commands in a separate thread. IPython has an internal job manager called jobs, and a convenience backgrounding magic function called '%bg'.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with '\$' is expanded. A double '\$\$' allows passing a literal '\$' to the shell (for access to shell and environment variables like PATH).
- Filesystem navigation, via a magic '%cd' command, along with a persistent bookmark system (using '%bookmark') for fast access to frequently visited directories.
- A lightweight persistence framework via the '%store' command, which allows you to save arbitrary Python variables. These get restored automatically when your session restarts.
- Automatic indentation (optional) of code as you type (through the readline library).
- Macro system for quickly re-executing multiple lines of previous input with a single name. Macros can be stored persistently via '%store' and edited via '%edit'.
- Session logging (you can then later use these logs as code in your programs). Logs can optionally timestamp all input, and also store session output (marked as comments, so the log remains valid Python source code).
- Session restoring: logs can be replayed to restore a previous session to the state where you left it.
- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information (basically a terminal version of the cgitb module).
- Auto-parentheses: callable objects can be executed without parentheses: 'sin 3' is automatically converted to 'sin(3)'.

- Auto-quoting: using `'`, `,` or `;` as the first character forces auto-quoting of the rest of the line: `',my_function a b'` becomes automatically `'my_function("a", "b")'`, while `';my_function a b'` becomes `'my_function("a b")'`.
- Extensible input syntax. You can define filters that pre-process user input to simplify input in special situations. This allows for example pasting multi-line code fragments which start with `>>>` or `...` such as those from other python sessions or the standard Python documentation.
- Flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.
- Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).
- Easy debugger access. You can set IPython to call up an enhanced version of the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The `%run` magic command (with the `-d` option) can run any script under pdb's control, automatically setting initial breakpoints for you. This version of pdb has IPython-specific improvements, including tab-completion and traceback coloring support. For even easier debugger access, try `%debug` after seeing an exception. winpdb is also supported, see `ipy_winpdb` extension.
- Profiler support. You can run single statements (similar to `profile.run()`) or complete programs under the profiler's control. While this is possible with standard `cProfile` or `profile` modules, IPython wraps this functionality with magic commands (see `%prun` and `%run -p`) convenient for rapid interactive work.
- Doctest support. The special `%doctest_mode` command toggles a mode that allows you to paste existing doctests (with leading `>>>` prompts and whitespace) and uses doctest-compatible prompts and output, so you can use IPython sessions as doctest code.

1.3 Interactive parallel computing

Increasingly, parallel computer hardware, such as multicore CPUs, clusters and supercomputers, is becoming ubiquitous. Over the last 3 years, we have developed an architecture within IPython that allows such hardware to be used quickly and easily from Python. Moreover, this architecture is designed to support interactive and collaborative parallel computing.

The main features of this system are:

- Quickly parallelize Python code from an interactive Python/IPython session.
- A flexible and dynamic process model that be deployed on anything from multicore workstations to supercomputers.
- An architecture that supports many different styles of parallelism, from message passing to task farming. And all of these styles can be handled interactively.
- Both blocking and fully asynchronous interfaces.

- High level APIs that enable many things to be parallelized in a few lines of code.
- Write parallel code that will run unchanged on everything from multicore workstations to supercomputers.
- Full integration with Message Passing libraries (MPI).
- Capabilities based security model with full encryption of network connections.
- Share live parallel jobs with other users securely. We call this collaborative parallel computing.
- Dynamically load balanced task farming system.
- Robust error handling. Python exceptions raised in parallel execution are gathered and presented to the top-level code.

For more information, see our [overview](#) of using IPython for parallel computing.

1.3.1 Portability and Python requirements

As of the 0.11 release, IPython works with either Python 2.5 or 2.6. Versions 0.9 and 0.10 worked with Python 2.4 as well. We have not yet begun to test and port IPython to Python 3. Our plan is to gradually drop Python 2.5 support and then begin the transition to strict 2.6 and 3.

IPython is known to work on the following operating systems:

- Linux
- Most other Unix-like OSs (AIX, Solaris, BSD, etc.)
- Mac OS X
- Windows (CygWin, XP, Vista, etc.)

See [here](#) for instructions on how to install IPython.

WHAT'S NEW IN IPYTHON

This section documents the changes that have been made in various versions of IPython. Users should consult these pages to learn about new features, bug fixes and backwards incompatibilities. Developers should summarize the development work they do here in a user friendly format.

2.1 Development version

2.1.1 Main *ipython* branch

As of the 0.11 version of IPython, a significant portion of the core has been refactored. This refactoring is founded on a number of new abstractions. The main new classes that implement these abstractions are:

- `IPython.utils.traitlets.HasTraitlets.`
- `IPython.core.component.Component.`
- `IPython.core.application.Application.`
- `IPython.config.loader.ConfigLoader.`
- `IPython.config.loader.Config`

We are still in the process of writing developer focused documentation about these classes, but for now our [configuration documentation](#) contains a high level overview of the concepts that these classes express.

The changes listed here are a brief summary of the recent work on IPython. For more details, please consult the actual source.

New features

- The `IPython.extensions.pretty` extension has been moved out of quarantine and fully updated to the new extension API.
- New magics for loading/unloading/reloading extensions have been added: `%load_ext`, `%unload_ext` and `%reload_ext`.
- The configuration system and configuration files are brand new. See the configuration system [documentation](#) for more details.

- The `InteractiveShell` class is now a `Component` subclass and has traitlets that determine the defaults and runtime environment. The `__init__` method has also been refactored so this class can be instantiated and run without the old `ipmaker` module.
- The methods of `InteractiveShell` have been organized into sections to make it easier to turn more sections of functionality into components.
- The embedded shell has been refactored into a truly standalone subclass of `InteractiveShell` called `InteractiveShellEmbed`. All embedding logic has been taken out of the base class and put into the embedded subclass.
- I have created methods of `InteractiveShell` to help it cleanup after itself. The `cleanup()` method controls this. We couldn't do this in `__del__()` because we have cycles in our object graph that prevent it from being called.
- Created a new module `IPython.utils.importstring` for resolving strings like `foo.bar.Bar` to the actual class.
- Completely refactored the `IPython.core.prefilter` module into `Component` subclasses. Added a new layer into the prefilter system, called “transformations” that all new prefilter logic should use (rather than the older “checker/handler” approach).
- Aliases are now components (`IPython.core.alias`).
- We are now using an internally shipped version of `argparse` to parse command line options for **ipython**.
- New top level `embed()` function that can be called to embed IPython at any place in user's code. One the first call it will create an `InteractiveShellEmbed` instance and call it. In later calls, it just calls the previously created `InteractiveShellEmbed`.
- Created a component system (`IPython.core.component`) that is based on `IPython.utils.traitlets`. Components are arranged into a runtime containment tree (not inheritance) that i) automatically propagates configuration information and ii) allows components to discover each other in a loosely coupled manner. In the future all parts of IPython will be subclasses of `Component`. All IPython developers should become familiar with the component system.
- Created a new `Config` for holding configuration information. This is a dict like class with a few extras: i) it supports attribute style access, ii) it has a merge function that merges two `Config` instances recursively and iii) it will automatically create sub-`Config` instances for attributes that start with an uppercase character.
- Created new configuration loaders in `IPython.config.loader`. These loaders provide a unified loading interface for all configuration information including command line arguments and configuration files. We have two default implementations based on `argparse` and plain python files. These are used to implement the new configuration system.
- Created a top-level `Application` class in `IPython.core.application` that is designed to encapsulate the starting of any IPython process. An application loads and merges all the configuration objects, constructs the main application `Component` instances and then starts the application running. The default `Application` class has built-in logic for handling the IPython directory as well as profiles.

- The `Type` and `Instance` traitlets now handle classes given as strings, like `foo.bar.Bar`. This is needed for forward declarations. But, this was implemented in a careful way so that string to class resolution is done at a single point, when the parent `HasTraitlets` is instantiated.
- `IPython.utils.ipstruct` has been refactored to be a subclass of `dict`. It also now has full docstrings and doctests.
- Created a Trait's like implementation in `IPython.utils.traitlets`. This is a pure Python, lightweight version of a library that is similar to `enthought.traits`. We are using this for validation, defaults and notification in our new component system. Although it is not API compatible with `enthought.traits`, we plan on moving in this direction so that eventually our implementation could be replaced by a (yet to exist) pure Python version of `enthought.traits`.
- Added a new module `IPython.lib.inputhook` to manage the integration with GUI event loops using *PyOS_InputHook*. See the docstrings in this module or the main IPython docs for details.
- For users, GUI event loop integration is now handled through the new `%gui` magic command. Type `%gui?` at an IPython prompt for documentation.
- The command line options `-wthread`, `-qthread` and `-gthread` just call the appropriate `IPython.lib.inputhook` functions.
- For developers `IPython.lib.inputhook` provides a simple interface for managing the event loops in their interactive GUI applications. Examples can be found in our `docs/examples/lib` directory.

Bug fixes

- Previously, the latex Sphinx docs were in a single chapter. This has been fixed by adding a sixth argument of `True` to the `latex_documents` attribute of `conf.py`.
- The `psum` example in the MPI documentation has been updated to `mpi4py` version 1.1.0. Thanks to J. Thomas for this fix.
- The top-level, zero-install `ipython.py` script has been updated to the new application launching API.
- Keyboard interrupts now work with GUI support enabled across all platforms and all GUI toolkits reliably.

Backwards incompatible changes

- The extension loading functions have been renamed to `load_ipython_extension()` and `unload_ipython_extension()`.
- `InteractiveShell` no longer takes an `embedded` argument. Instead just use the `InteractiveShellEmbed` class.
- `__IPYTHON__` is no longer injected into `__builtin__`.
- `Struct.__init__()` no longer takes `None` as its first argument. It must be a `dict` or `Struct`.
- `ipmagic()` has been renamed `()`

- The functions `ipmagic()` and `ipalias()` have been removed from `__builtins__`.
- The references to the global `InteractiveShell` instance (`_ip`, and `__IP`) have been removed from the user's namespace. They are replaced by a new function called `get_ipython()` that returns the current `InteractiveShell` instance. This function is injected into the user's namespace and is now the main way of accessing IPython's API.
- Old style configuration files `ipythonrc` and `ipy_user_conf.py` are no longer supported. Users should migrate there configuration files to the new format described [here](#) and [here](#).
- The old IPython extension API that relied on `ipapi()` has been completely removed. The new extension API is described [here](#).
- Support for `qt3` has been dropped. User's who need this should use previous versions of IPython.
- Removed `shellglobals` as it was obsolete.
- Removed all the threaded shells in `IPython.core.shell`. These are no longer needed because of the new capabilities in `IPython.lib.inputhook`.
- The `-pylab` command line flag has been disabled until `matplotlib` adds support for the new `IPython.lib.inputhook` approach. The new stuff does work with `matplotlib`, but you have to set everything up by hand.
- New top-level sub-packages have been created: `IPython.core`, `IPython.lib`, `IPython.utils`, `IPython.deathrow`, `IPython.quarantine`. All existing top-level modules have been moved to appropriate sub-packages. All internal import statements have been updated and tests have been added. The build system (`setup.py` and `friends`) have been updated. See [this section](#) of the documentation for descriptions of these new sub-packages.
- Compatibility modules have been created for `IPython.Shell`, `IPython.ipapi` and `IPython.ipilib` that display warnings and then load the actual implementation from `IPython.core`.
- `Extensions` has been moved to `extensions` and all existing extensions have been moved to either `IPython.quarantine` or `IPython.deathrow`. `IPython.quarantine` contains modules that we plan on keeping but that need to be updated. `IPython.deathrow` contains modules that are either dead or that should be maintained as third party libraries. More details about this can be found [here](#).
- The IPython GUIs in `IPython.frontend` and `IPython.gui` are likely broken because of the refactoring in the core. With proper updates, these should still work. We probably want to get these so they are not using `IPython.kernel.core` (which is being phased out).

2.2 0.10 series

2.2.1 Release 0.10.1

IPython 0.10.1 was released October 11, 2010, over a year after version 0.10. This is mostly a bugfix release, since after version 0.10 was released, the development team's energy has been focused on the 0.11 series. We have nonetheless tried to backport what fixes we could into 0.10.1, as it remains the stable series that many users have in production systems they rely on.

Since the 0.11 series changes many APIs in backwards-incompatible ways, we are willing to continue maintaining the 0.10.x series. We don't really have time to actively write new code for 0.10.x, but we are happy to accept patches and pull requests on the IPython [github site](#). If sufficient contributions are made that improve 0.10.1, we will roll them into future releases. For this purpose, we will have a branch called 0.10.2 on github, on which you can base your contributions.

For this release, we applied approximately 60 commits totaling a diff of over 7000 lines:

```
(0.10.1)amirbar[dist]> git diff --oneline rel-0.10.. | wc -l
7296
```

Highlights of this release:

- The only significant new feature is that IPython's parallel computing machinery now supports natively the Sun Grid Engine and LSF schedulers. This work was a joint contribution from Justin Riley, Satra Ghosh and Matthieu Brucher, who put a lot of work into it. We also improved traceback handling in remote tasks, as well as providing better control for remote task IDs.
- New IPython Sphinx directive contributed by John Hunter. You can use this directive to mark blocks in reStructuredText documents as containing IPython syntax (including figures) and the will be executed during the build:

```
In [2]: plt.figure() # ensure a fresh figure

@savefig psimple.png width=4in
In [3]: plt.plot([1,2,3])
Out[3]: [<matplotlib.lines.Line2D object at 0x9b74d8c>]
```

- Various fixes to the standalone ipython-wx application.
- We now ship internally the excellent argparse library, graciously licensed under BSD terms by Steven Bethard. Now (2010) that argparse has become part of Python 2.7 this will be less of an issue, but Steven's relicensing allowed us to start updating IPython to using argparse well before Python 2.7. Many thanks!
- Robustness improvements so that IPython doesn't crash if the readline library is absent (though obviously a lot of functionality that requires readline will not be available).
- Improvements to tab completion in Emacs with Python 2.6.
- Logging now supports timestamps (see `%logstart?` for full details).
- A long-standing and quite annoying bug where parentheses would be added to `print` statements, under Python 2.5 and 2.6, was finally fixed.
- Improved handling of libreadline on Apple OSX.
- Fix `reload` method of IPython demos, which was broken.
- Fixes for the ipipe/ibrowse system on OSX.
- Fixes for Zope profile.
- Fix `%timeit` reporting when the time is longer than 1000s.
- Avoid lockups with `?` or `??` in SunOS, due to a bug in termios.

- The usual assortment of miscellaneous bug fixes and small improvements.

The following people contributed to this release (please let us know if we omitted your name and we'll gladly fix this in the notes for the future):

- Beni Cherniavsky
- Boyd Waters.
- David Warde-Farley
- Fernando Perez
- Gökhan Sever
- John Hunter
- Justin Riley
- Kiorky
- Laurent Dufrechou
- Mark E. Smith
- Matthieu Brucher
- Satrajit Ghosh
- Sebastian Busch
- Václav Šmilauer

2.2.2 Release 0.10

This release brings months of slow but steady development, and will be the last before a major restructuring and cleanup of IPython's internals that is already under way. For this reason, we hope that 0.10 will be a stable and robust release so that while users adapt to some of the API changes that will come with the refactoring that will become IPython 0.11, they can safely use 0.10 in all existing projects with minimal changes (if any).

IPython 0.10 is now a medium-sized project, with roughly (as reported by David Wheeler's **sloccount** utility) 40750 lines of Python code, and a diff between 0.9.1 and this release that contains almost 28000 lines of code and documentation. Our documentation, in PDF format, is a 495-page long PDF document (also available in HTML format, both generated from the same sources).

Many users and developers contributed code, features, bug reports and ideas to this release. Please do not hesitate in contacting us if we've failed to acknowledge your contribution here. In particular, for this release we have contribution from the following people, a mix of new and regular names (in alphabetical order by first name):

- Alexander Clausen: fix #341726.
- Brian Granger: lots of work everywhere (features, bug fixes, etc).
- Daniel Ashbrook: bug report on MemoryError during compilation, now fixed.

- Darren Dale: improvements to documentation build system, feedback, design ideas.
- Fernando Perez: various places.
- Gaël Varoquaux: core code, ipythonx GUI, design discussions, etc. Lots...
- John Hunter: suggestions, bug fixes, feedback.
- Jorgen Stenarson: work on many fronts, tests, fixes, win32 support, etc.
- Laurent Dufréhou: many improvements to ipython-wx standalone app.
- Lukasz Pankowski: prefilter, `%edit`, demo improvements.
- Matt Foster: TextMate support in `%edit`.
- Nathaniel Smith: fix #237073.
- Pauli Virtanen: fixes and improvements to extensions, documentation.
- Prabhu Ramachandran: improvements to `%timeit`.
- Robert Kern: several extensions.
- Sameer D'Costa: help on critical bug #269966.
- Stephan Peijnik: feedback on Debian compliance and many man pages.
- Steven Bethard: we are now shipping his `argparse` module.
- Tom Fetherston: many improvements to `IPython.demos` module.
- Ville Vainio: lots of work everywhere (features, bug fixes, etc).
- Vishal Vasta: ssh support in `ipcluster`.
- Walter Doerwald: work on the `IPython.ipipe` system.

Below we give an overview of new features, bug fixes and backwards-incompatible changes. For a detailed account of every change made, feel free to view the project log with **bzr log**.

New features

- New `%paste` magic automatically extracts current contents of clipboard and pastes it directly, while correctly handling code that is indented or prepended with `>>>` or `...` python prompt markers. A very useful new feature contributed by Robert Kern.
- IPython 'demos', created with the `IPython.demos` module, can now be created from files on disk or strings in memory. Other fixes and improvements to the demo system, by Tom Fetherston.
- Added `find_cmd()` function to `IPython.platutils` module, to find commands in a cross-platform manner.
- Many improvements and fixes to Gaël Varoquaux's **ipythonx**, a WX-based lightweight IPython instance that can be easily embedded in other WX applications. These improvements have made it possible to now have an embedded IPython in Mayavi and other tools.
- `MultiengineClient` objects now have a `benchmark()` method.

- The manual now includes a full set of auto-generated API documents from the code sources, using Sphinx and some of our own support code. We are now using the [Numpy Documentation Standard](#) for all docstrings, and we have tried to update as many existing ones as possible to this format.
- The new `IPython.Extensions.ipy_pretty` extension by Robert Kern provides configurable pretty-printing.
- Many improvements to the **ipython-wx** standalone WX-based IPython application by Laurent Dufr  chou. It can optionally run in a thread, and this can be toggled at runtime (allowing the loading of Matplotlib in a running session without ill effects).
- IPython includes a copy of Steven Bethard's [argparse](#) in the `IPython.external` package, so we can use it internally and it is also available to any IPython user. By installing it in this manner, we ensure zero conflicts with any system-wide installation you may already have while minimizing external dependencies for new users. In IPython 0.10, We ship argparse version 1.0.
- An improved and much more robust test suite, that runs groups of tests in separate subprocesses using either Nose or Twisted's **trial** runner to ensure proper management of Twisted-using code. The test suite degrades gracefully if optional dependencies are not available, so that the **iptest** command can be run with only Nose installed and nothing else. We also have more and cleaner test decorators to better select tests depending on runtime conditions, do setup/teardown, etc.
- The new `ipcluster` now has a fully working ssh mode that should work on Linux, Unix and OS X. Thanks to Vishal Vatsa for implementing this!
- The wonderful TextMate editor can now be used with `%edit` on OS X. Thanks to Matt Foster for this patch.
- The documentation regarding parallel uses of IPython, including MPI and PBS, has been significantly updated and improved.
- The developer guidelines in the documentation have been updated to explain our workflow using **bzr** and Launchpad.
- Fully refactored **ipcluster** command line program for starting IPython clusters. This new version is a complete rewrite and 1) is fully cross platform (we now use Twisted's process management), 2) has much improved performance, 3) uses subcommands for different types of clusters, 4) uses argparse for parsing command line options, 5) has better support for starting clusters using **mpirun**, 6) has experimental support for starting engines using PBS. It can also reuse FURL files, by appropriately passing options to its subcommands. However, this new version of `ipcluster` should be considered a technology preview. We plan on changing the API in significant ways before it is final.
- Full description of the security model added to the docs.
- `cd` completer: show bookmarks if no other completions are available.
- `sh` profile: easy way to give 'title' to prompt: assign to variable `'_prompt_title'`. It looks like this:

```
[~]|1> _prompt_title = 'sudo!'
sudo! [~]|2>
```
- `%edit`: If you do `'%edit pasted_block'`, `pasted_block` variable gets updated with new data (so repeated editing makes sense)

Bug fixes

- Fix #368719, removed top-level debian/ directory to make the job of Debian packagers easier.
- Fix #291143 by including man pages contributed by Stephan Peijnik from the Debian project.
- Fix #358202, effectively a race condition, by properly synchronizing file creation at cluster startup time.
- `%timeit` now handles correctly functions that take a long time to execute even the first time, by not repeating them.
- Fix #239054, releasing of references after exiting.
- Fix #341726, thanks to Alexander Clausen.
- Fix #269966. This long-standing and very difficult bug (which is actually a problem in Python itself) meant long-running sessions would inevitably grow in memory size, often with catastrophic consequences if users had large objects in their scripts. Now, using `%run` repeatedly should not cause any memory leaks. Special thanks to John Hunter and Sameer D'Costa for their help with this bug.
- Fix #295371, bug in `%history`.
- Improved support for py2exe.
- Fix #270856: IPython hangs with PyGTK
- Fix #270998: A magic with no docstring breaks the ‘`%magic magic`’
- fix #271684: -c startup commands screw up raw vs. native history
- Numerous bugs on Windows with the new ipcluster have been fixed.
- The ipengine and ipcontroller scripts now handle missing furl files more gracefully by giving better error messages.
- `%rehashx`: Aliases no longer contain dots. python3.0 binary will create alias python30. Fixes: #259716 “commands with dots in them don’t work”
- `%cpaste`: `%cpaste -r` repeats the last pasted block. The block is assigned to `pasted_block` even if code raises exception.
- Bug #274067 ‘The code in `get_home_dir` is broken for py2exe’ was fixed.
- Many other small bug fixes not listed here by number (see the bzt log for more info).

Backwards incompatible changes

- `ipykit` and related files were unmaintained and have been removed.
- The `IPython.genutils.doctest_reload()` does not actually call `reload(doctest)` anymore, as this was causing many problems with the test suite. It still resets `doctest.master` to `None`.
- While we have not deliberately broken Python 2.4 compatibility, only minor testing was done with Python 2.4, while 2.5 and 2.6 were fully tested. But if you encounter problems with 2.4, please do report them as bugs.

- The **ipcluster** now requires a mode argument; for example to start a cluster on the local machine with 4 engines, you must now type:

```
$ ipcluster local -n 4
```

- The controller now has a `-r` flag that needs to be used if you want to reuse existing furl files. Otherwise they are deleted (the default).
- Remove `ipy_leo.py`. You can use **easy_install ipython-extension** to get it. (done to decouple it from ipython release cycle)

2.3 0.9 series

2.3.1 Release 0.9.1

This release was quickly made to restore compatibility with Python 2.4, which version 0.9 accidentally broke. No new features were introduced, other than some additional testing support for internal use.

2.3.2 Release 0.9

New features

- All furl files and security certificates are now put in a read-only directory named `~/ipython/security`.
- A single function `get_ipython_dir()`, in `IPython.genutils` that determines the user's IPython directory in a robust manner.
- Laurent's WX application has been given a top-level script called `ipython-wx`, and it has received numerous fixes. We expect this code to be architecturally better integrated with Gael's WX 'ipython widget' over the next few releases.
- The Editor synchronization work by Vivian De Smedt has been merged in. This code adds a number of new editor hooks to synchronize with editors under Windows.
- A new, still experimental but highly functional, WX shell by Gael Varoquaux. This work was sponsored by Enthought, and while it's still very new, it is based on a more cleanly organized architecture of the various IPython components. We will continue to develop this over the next few releases as a model for GUI components that use IPython.
- Another GUI frontend, Cocoa based (Cocoa is the OSX native GUI framework), authored by Barry Wark. Currently the WX and the Cocoa ones have slightly different internal organizations, but the whole team is working on finding what the right abstraction points are for a unified codebase.
- As part of the frontend work, Barry Wark also implemented an experimental event notification system that various ipython components can use. In the next release the implications and use patterns of this system regarding the various GUI options will be worked out.
- IPython finally has a full test system, that can test docstrings with IPython-specific functionality. There are still a few pieces missing for it to be widely accessible to all users (so they can run the test suite at any time and report problems), but it now works for the developers. We are working hard on

continuing to improve it, as this was probably IPython's major Achilles heel (the lack of proper test coverage made it effectively impossible to do large-scale refactoring). The full test suite can now be run using the **iptest** command line program.

- The notion of a task has been completely reworked. An *ITask* interface has been created. This interface defines the methods that tasks need to implement. These methods are now responsible for things like submitting tasks and processing results. There are two basic task types: `IPython.kernel.task.StringTask` (this is the old *Task* object, but renamed) and the new `IPython.kernel.task.MapTask`, which is based on a function.
- A new interface, `IPython.kernel.mapper.IMapper` has been defined to standardize the idea of a *map* method. This interface has a single *map* method that has the same syntax as the built-in *map*. We have also defined a *mapper* factory interface that creates objects that implement `IPython.kernel.mapper.IMapper` for different controllers. Both the multiengine and task controller now have mapping capabilities.
- The parallel function capabilities have been reworks. The major changes are that i) there is now an *@parallel* magic that creates parallel functions, ii) the syntax for multiple variable follows that of *map*, iii) both the multiengine and task controller now have a parallel function implementation.
- All of the parallel computing capabilities from *ipython1-dev* have been merged into IPython proper. This resulted in the following new subpackages: `IPython.kernel`, `IPython.kernel.core`, `IPython.config`, `IPython.tools` and `IPython.testing`.
- As part of merging in the *ipython1-dev* stuff, the *setup.py* script and friends have been completely refactored. Now we are checking for dependencies using the approach that matplotlib uses.
- The documentation has been completely reorganized to accept the documentation from *ipython1-dev*.
- We have switched to using Foolscap for all of our network protocols in `IPython.kernel`. This gives us secure connections that are both encrypted and authenticated.
- We have a brand new *COPYING.txt* files that describes the IPython license and copyright. The biggest change is that we are putting "The IPython Development Team" as the copyright holder. We give more details about exactly what this means in this file. All developer should read this and use the new banner in all IPython source code files.
- sh profile: `./foo` runs `foo` as system command, no need to do `!./foo` anymore
- String lists now support `sort(field, nums = True)` method (to easily sort system command output). Try it with `a = !ls -l ; a.sort(1, nums=1)`.
- `%cpaste foo` now assigns the pasted block as string list, instead of string
- The *ipcluster* script now run by default with no security. This is done because the main usage of the script is for starting things on localhost. Eventually when *ipcluster* is able to start things on other hosts, we will put security back.
- `'cd -foo'` searches directory history for string `foo`, and jumps to that dir. Last part of dir name is checked first. If no matches for that are found, look at the whole path.

Bug fixes

- The Windows installer has been fixed. Now all IPython scripts have `.bat` versions created. Also, the Start Menu shortcuts have been updated.
- The colors escapes in the multiengine client are now turned off on win32 as they don't print correctly.
- The `IPython.kernel.scripts.ipengine` script was exec'ing `mpi_import_statement` incorrectly, which was leading the engine to crash when mpi was enabled.
- A few subpackages had missing `__init__.py` files.
- The documentation is only created if Sphinx is found. Previously, the `setup.py` script would fail if it was missing.
- Greedy `cd` completion has been disabled again (it was enabled in 0.8.4) as it caused problems on certain platforms.

Backwards incompatible changes

- The `clusterfile` options of the **ipcluster** command has been removed as it was not working and it will be replaced soon by something much more robust.
- The `IPython.kernel` configuration now properly find the user's IPython directory.
- In `ipapi`, the `make_user_ns()` function has been replaced with `make_user_namespaces()`, to support dict subclasses in namespace creation.
- `IPython.kernel.client.Task` has been renamed `IPython.kernel.client.StringTask` to make way for new task types.
- The keyword argument `style` has been renamed `dist` in `scatter`, `gather` and `map`.
- Renamed the values that the rename `dist` keyword argument can have from `'basic'` to `'b'`.
- IPython has a larger set of dependencies if you want all of its capabilities. See the `setup.py` script for details.
- The constructors for `IPython.kernel.client.MultiEngineClient` and `IPython.kernel.client.TaskClient` no longer take the `(ip,port)` tuple. Instead they take the filename of a file that contains the FURL for that client. If the FURL file is in your `IPYTHONDIR`, it will be found automatically and the constructor can be left empty.
- The asynchronous clients in `IPython.kernel.asyncclient` are now created using the factory functions `get_multiengine_client()` and `get_task_client()`. These return a *Deferred* to the actual client.
- The command line options to `ipcontroller` and `ipengine` have changed to reflect the new Foolsmap network protocol and the FURL files. Please see the help for these scripts for details.
- The configuration files for the kernel have changed because of the Foolsmap stuff. If you were using custom config files before, you should delete them and regenerate new ones.

Changes merged in from IPython1

New features

- Much improved `setup.py` and `setuptools.py` scripts. Because Twisted and `zope.interface` are now easy installable, we can declare them as dependencies in our `setuptools.py` script.
- IPython is now compatible with Twisted 2.5.0 and 8.x.
- Added a new example of how to use `ipython1.kernel.asyncclient`.
- Initial draft of a process daemon in `ipython1.daemon`. This has not been merged into IPython and is still in *ipython1-dev*.
- The `TaskController` now has methods for getting the queue status.
- The `TaskResult` objects now have information about how long the task took to run.
- We are attaching additional attributes to exceptions (`_ipython_*`) that we use to carry additional info around.
- New top-level module `asyncclient` that has asynchronous versions (that return deferreds) of the client classes. This is designed to users who want to run their own Twisted reactor.
- All the clients in `client` are now based on Twisted. This is done by running the Twisted reactor in a separate thread and using the `blockingCallFromThread()` function that is in recent versions of Twisted.
- Functions can now be pushed/pulled to/from engines using `MultiEngineClient.push_function()` and `MultiEngineClient.pull_function()`.
- Gather/scatter are now implemented in the client to reduce the work load of the controller and improve performance.
- Complete rewrite of the IPython documentation. All of the documentation from the IPython website has been moved into `docs/source` as restructured text documents. PDF and HTML documentation are being generated using Sphinx.
- New developer oriented documentation: development guidelines and roadmap.
- Traditional `ChangeLog` has been changed to a more useful `changes.txt` file that is organized by release and is meant to provide something more relevant for users.

Bug fixes

- Created a proper `MANIFEST.in` file to create source distributions.
- Fixed a bug in the `MultiEngine` interface. Previously, multi-engine actions were being collected with a `DeferredList` with `fireononeerrback=1`. This meant that methods were returning before all engines had given their results. This was causing extremely odd bugs in certain cases. To fix this problem, we have 1) set `fireononeerrback=0` to make sure all results (or exceptions) are in before returning and 2) introduced a `CompositeError` exception that wraps all of the engine

exceptions. This is a huge change as it means that users will have to catch `CompositeError` rather than the actual exception.

Backwards incompatible changes

- All names have been renamed to conform to the `lowercase_with_underscore` convention. This will require users to change references to all names like `queueStatus` to `queue_status`.
- Previously, methods like `MultiEngineClient.push()` and `MultiEngineClient.push()` used `*args` and `**kwargs`. This was becoming a problem as we weren't able to introduce new keyword arguments into the API. Now these methods simply take a dict or sequence. This has also allowed us to get rid of the `*All` methods like `pushAll()` and `pullAll()`. These things are now handled with the `targets` keyword argument that defaults to `'all'`.
- The `MultiEngineClient.magicTargets` has been renamed to `MultiEngineClient.targets`.
- All methods in the `MultiEngine` interface now accept the optional keyword argument `block`.
- Renamed `RemoteController` to `MultiEngineClient` and `TaskController` to `TaskClient`.
- Renamed the top-level module from `api` to `client`.
- Most methods in the `multiengine` interface now raise a `CompositeError` exception that wraps the user's exceptions, rather than just raising the raw user's exception.
- Changed the `setupNS` and `resultNames` in the `Task` class to `push` and `pull`.

2.4 0.8 series

2.4.1 Release 0.8.4

This was a quick release to fix an unfortunate bug that slipped into the 0.8.3 release. The `--twisted` option was disabled, as it turned out to be broken across several platforms.

2.4.2 Release 0.8.3

- `pydb` is now disabled by default (due to `%run -d` problems). You can enable it by passing `-pydb` command line argument to IPython. Note that setting it in config file won't work.

2.4.3 Release 0.8.2

- `%pushd/%popd` behave differently; now `"pushd /foo"` pushes `CURRENT` directory and jumps to `/foo`. The current behaviour is closer to the documented behaviour, and should not trip anyone.

2.4.4 Older releases

Changes in earlier releases of IPython are described in the older file `ChangeLog`. Please refer to this document for details.

INSTALLATION

3.1 Overview

This document describes the steps required to install IPython. IPython is organized into a number of sub-packages, each of which has its own dependencies. All of the subpackages come with IPython, so you don't need to download and install them separately. However, to use a given subpackage, you will need to install all of its dependencies.

Please let us know if you have problems installing IPython or any of its dependencies. Officially, IPython requires Python version 2.5 or 2.6. We have *not* yet started to port IPython to Python 3.0.

Warning: Officially, IPython supports Python versions 2.5 and 2.6. IPython 0.10 has only been well tested with Python 2.5 and 2.6. Parts of it may work with Python 2.4, but we do not officially support Python 2.4 anymore. If you need to use 2.4, you can still run IPython 0.9.

Some of the installation approaches use the `setuptools` package and its `easy_install` command line program. In many scenarios, this provides the most simple method of installing IPython and its dependencies. It is not required though. More information about `setuptools` can be found on its website.

More general information about installing Python packages can be found in Python's documentation at <http://www.python.org/doc/>.

3.2 Quickstart

If you have `setuptools` installed and you are on OS X or Linux (not Windows), the following will download and install IPython *and* the main optional dependencies:

```
$ easy_install ipython[kernel,security,test]
```

This will get Twisted, `zope.interface` and `Foolscap`, which are needed for IPython's parallel computing features as well as the `nose` package, which will enable you to run IPython's test suite.

Warning: IPython's test system is being refactored and currently the `iptest` shown below does not work. More details about the testing situation can be found [here](#)

To run IPython's test suite, use the **iptest** command:

```
$ iptest
```

Read on for more specific details and instructions for Windows.

3.3 Installing IPython itself

Given a properly built Python, the basic interactive IPython shell will work with no external dependencies. However, some Python distributions (particularly on Windows and OS X), don't come with a working `readline` module. The IPython shell will work without `readline`, but will lack many features that users depend on, such as tab completion and command line editing. See below for details of how to make sure you have a working `readline`.

3.3.1 Installation using `easy_install`

If you have `setuptools` installed, the easiest way of getting IPython is to simply use **easy_install**:

```
$ easy_install ipython
```

That's it.

3.3.2 Installation from source

If you don't want to use **easy_install**, or don't have it installed, just grab the latest stable build of IPython from [here](#). Then do the following:

```
$ tar -xzf ipython.tar.gz
$ cd ipython
$ python setup.py install
```

If you are installing to a location (like `/usr/local`) that requires higher permissions, you may need to run the last command with **sudo**.

3.3.3 Windows

There are a few caveats for Windows users. The main issue is that a basic `python setup.py install` approach won't create `.bat` file or Start Menu shortcuts, which most users want. To get an installation with these, you can use any of the following alternatives:

1. Install using **easy_install**.
2. Install using our binary `.exe` Windows installer, which can be found at [here](#)
3. Install from source, but using `setuptools` (`python setupegg.py install`).

IPython by default runs in a terminal window, but the normal terminal application supplied by Microsoft Windows is very primitive. You may want to download the excellent and free [Console](#) application instead,

which is a far superior tool. You can even configure Console to give you by default an IPython tab, which is very convenient to create new IPython sessions directly from the working terminal.

Note for Windows 64 bit users: you may have difficulties with the stock installer on 64 bit systems; in this case (since we currently do not have 64 bit builds of the Windows installer) your best bet is to install from source with the `setuptools` method indicated in #3 above. See [this bug report](#) for further details.

3.3.4 Installing the development version

It is also possible to install the development version of IPython from our [Bazaar](#) source code repository. To do this you will need to have Bazaar installed on your system. Then just do:

```
$ bazaar branch lp:ipython
$ cd ipython
$ python setup.py install
```

Again, this last step on Windows won't create `.bat` files or Start Menu shortcuts, so you will have to use one of the other approaches listed above.

Some users want to be able to follow the development branch as it changes. If you have `setuptools` installed, this is easy. Simply replace the last step by:

```
$ python setupegg.py develop
```

This creates links in the right places and installs the command line script to the appropriate places. Then, if you want to update your IPython at any time, just do:

```
$ bazaar pull
```

3.4 Basic optional dependencies

There are a number of basic optional dependencies that most users will want to get. These are:

- `readline` (for command line editing, tab completion, etc.)
- `nose` (to run the IPython test suite)
- `pexpect` (to use things like `irunner`)

If you are comfortable installing these things yourself, have at it, otherwise read on for more details.

3.4.1 readline

In principle, all Python distributions should come with a working `readline` module. But, reality is not quite that simple. There are two common situations where you won't have a working `readline` module:

- If you are using the built-in Python on Mac OS X.
- If you are running Windows, which doesn't have a `readline` module.

On OS X, the built-in Python doesn't not have `readline` because of license issues. Starting with OS X 10.5 (Leopard), Apple's built-in Python has a BSD-licensed not-quite-compatible `readline` replacement. As of IPython 0.9, many of the issues related to the differences between `readline` and `libedit` seem to have been resolved. While you may find `libedit` sufficient, we have occasional reports of bugs with it and several developers who use OS X as their main environment consider `libedit` unacceptable for productive, regular use with IPython.

Therefore, we *strongly* recommend that on OS X you get the full `readline` module. We will *not* consider completion/history problems to be bugs for IPython if you are using `libedit`.

To get a working `readline` module, just do (with `setuptools` installed):

```
$ easy_install readline
```

Note: Other Python distributions on OS X (such as `fink`, `MacPorts` and the official `python.org` binaries) already have `readline` installed so you likely don't have to do this step.

If needed, the `readline` egg can be build and installed from source (see the wiki page at <http://ipython.scipy.org/moin/InstallationOSXLeopard>).

On Windows, you will need the `PyReadline` module. `PyReadline` is a separate, Windows only implementation of `readline` that uses native Windows calls through `ctypes`. The easiest way of installing `PyReadline` is you use the binary installer available [here](#). The `ctypes` module, which comes with Python 2.5 and greater, is required by `PyReadline`. It is available for Python 2.4 at <http://python.net/crew/theller/ctypes>.

3.4.2 nose

To run the IPython test suite you will need the `nose` package. `Nose` provides a great way of sniffing out and running all of the IPython tests. The simplest way of getting `nose`, is to use **easy_install**:

```
$ easy_install nose
```

Another way of getting this is to do:

```
$ easy_install ipython[test]
```

For more installation options, see the [nose website](#).

Warning: As described above, the `iptest` command currently doesn't work.

Once you have `nose` installed, you can run IPython's test suite using the `iptest` command:

```
$ iptest
```

3.4.3 pexpect

The `pexpect` package is used in IPython's `irunner` script. On Unix platforms (including OS X), just do:


```
$ easy_install pexpect
```

Windows users are out of luck as pexpect does not run there.

3.5 Dependencies for IPython.kernel (parallel computing)

The IPython kernel provides a nice architecture for parallel computing. The main focus of this architecture is on interactive parallel computing. These features require a number of additional packages:

- `zope.interface` (yep, we use interfaces)
- Twisted (asynchronous networking framework)
- Foolscape (a nice, secure network protocol)
- `pyOpenSSL` (security for network connections)

On a Unix style platform (including OS X), if you want to use `setuptools`, you can just do:

```
$ easy_install ipython[kernel]      # the first three
$ easy_install ipython[security]    # pyOpenSSL
```

3.5.1 `zope.interface` and Twisted

Twisted [Twisted] and `zope.interface` [ZopeInterface] are used for networking related things. On Unix style platforms (including OS X), the simplest way of getting these is to use **easy_install**:

```
$ easy_install zope.interface
$ easy_install Twisted
```

Of course, you can also download the source tarballs from the Twisted website [Twisted] and the `zope.interface` page at PyPI and do the usual `python setup.py install` if you prefer.

Windows is a bit different. For `zope.interface` and Twisted, simply get the latest binary `.exe` installer from the Twisted website. This installer includes both `zope.interface` and Twisted and should just work.

3.5.2 Foolscape

Foolscape [Foolscape] uses Twisted to provide a very nice secure RPC protocol that we use to implement our parallel computing features.

On all platforms a simple:

```
$ easy_install foolscap
```

should work. You can also download the source tarballs from the Foolscape website and do `python setup.py install` if you prefer.

3.5.3 pyOpenSSL

IPython does not work with version 0.7 of pyOpenSSL [\[pyOpenSSL\]](#). It is known to work with version 0.6 and will likely work with the more recent 0.8 and 0.9 versions. There are a couple of options for getting this:

1. Most Linux distributions have packages for pyOpenSSL.
2. The built-in Python 2.5 on OS X 10.5 already has it installed.
3. There are source tarballs on the pyOpenSSL website. On Unix-like platforms, these can be built using `python setup.py install`.
4. There is also a binary `.exe` Windows installer on the [pyOpenSSL website](#).

3.6 Dependencies for IPython.frontend (the IPython GUI)

3.6.1 wxPython

Starting with IPython 0.9, IPython has a new `IPython.frontend` package that has a nice wxPython based IPython GUI. As you would expect, this GUI requires wxPython. Most Linux distributions have wxPython packages available and the built-in Python on OS X comes with wxPython preinstalled. For Windows, a binary installer is available on the [wxPython website](#).

USING IPYTHON FOR INTERACTIVE WORK

4.1 Quick IPython tutorial

Warning: As of the 0.11 version of IPython, some of the features and APIs described in this section have been deprecated or are broken. Our plan is to continue to support these features, but they need to be updated to take advantage of recent API changes. Furthermore, this section of the documentation need to be updated to reflect all of these changes.

IPython can be used as an improved replacement for the Python prompt, and for that you don't really need to read any more of this manual. But in this section we'll try to summarize a few tips on how to make the most effective use of it for everyday Python development, highlighting things you might miss in the rest of the manual (which is getting long). We'll give references to parts in the manual which provide more detail when appropriate.

The following article by Jeremy Jones provides an introductory tutorial about IPython: <http://www.onlamp.com/pub/a/python/2005/01/27/ipython.html>

4.1.1 Highlights

Tab completion

TAB-completion, especially for attributes, is a convenient way to explore the structure of any object you're dealing with. Simply type `object_name.<TAB>` and a list of the object's attributes will be printed (see [the readline section](#) for more). Tab completion also works on file and directory names, which combined with IPython's alias system allows you to do from within IPython many of the things you normally would need the system shell for.

Explore your objects

Typing `object_name?` will print all sorts of details about any object, including docstrings, function definition lines (for call arguments) and constructor details for classes. The magic commands `%pdoc`, `%pdef`,

`%psource` and `%pfile` will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found). If automagic is on (it is by default), you don't need to type the `'%'` explicitly. See [this section](#) for more.

The `%run` magic command

The `%run` magic command allows you to run any python script and load all of its data directly into the interactive namespace. Since the file is re-read from disk each time, changes you make to it are reflected immediately (in contrast to the behavior of `import`). I rarely use `import` for code I am testing, relying on `%run` instead. See [this section](#) for more on this and other magic commands, or type the name of any magic command and `?` to get details on it. See also [this section](#) for a recursive reload command. `%run` also has special flags for timing the execution of your scripts (`-t`) and for executing them under the control of either Python's `pdb` debugger (`-d`) or profiler (`-p`). With all of these, `%run` can be used as the main tool for efficient interactive development of code which you write in your editor of choice.

Debug a Python script

Use the Python debugger, `pdb`. The `%pdb` command allows you to toggle on and off the automatic invocation of an IPython-enhanced `pdb` debugger (with coloring, tab completion and more) at any uncaught exception. The advantage of this is that `pdb` starts inside the function where the exception occurred, with all data still available. You can print variables, see code, execute statements and even walk up and down the call stack to track down the true source of the problem (which often is many layers in the stack above where the exception gets triggered). Running programs with `%run` and `pdb` active can be an efficient to develop and debug code, in many cases eliminating the need for print statements or external debugging tools. I often simply put a `1/0` in a place where I want to take a look so that `pdb` gets called, quickly view whatever variables I need to or test various pieces of code and then remove the `1/0`. Note also that `'%run -d'` activates `pdb` and automatically sets initial breakpoints for you to step through your code, watch variables, etc. The [output caching section](#) has more details.

Use the output cache

All output results are automatically stored in a global dictionary named `Out` and variables named `_1`, `_2`, etc. alias them. For example, the result of input line 4 is available either as `Out[4]` or as `_4`. Additionally, three variables named `_`, `__` and `___` are always kept updated with the for the last three results. This allows you to recall any previous result and further use it for new calculations. See [the output caching section](#) for more.

Suppress output

Put a `';`' at the end of a line to suppress the printing of output. This is useful when doing calculations which generate long output you are not interested in seeing. The `_*` variables and the `Out[]` list do get updated with the contents of the output, even if it is not printed. You can thus still access the generated results this way for further processing.

Input cache

A similar system exists for caching input. All input is stored in a global list called `In`, so you can re-execute lines 22 through 28 plus line 34 by typing `'exec In[22:29]+In[34]'` (using Python slicing notation). If you need to execute the same set of lines often, you can assign them to a macro with the `%macro` function. See [here](#) for more.

Use your input history

The `%hist` command can show you all previous input, without line numbers if desired (option `-n`) so you can directly copy and paste code either back in IPython or in a text editor. You can also save all your history by turning on logging via `%logstart`; these logs can later be either reloaded as IPython sessions or used as code for your programs.

In particular, note that the `%rep` magic function can repeat a command or get a command to the input line for further editing:

```
$ l = ["hei", "vaan"]
$ "".join(l)
==> heivaan
$ %rep
$ heivaan_ <== cursor blinking
```

For more details, type `%rep?` as usual.

Define your own system aliases

Even though IPython gives you access to your system shell via the `!` prefix, it is convenient to have aliases to the system commands you use most often. This allows you to work seamlessly from inside IPython with the same commands you are used to in your system shell. IPython comes with some pre-defined aliases and a complete system for changing directories, both via a stack (see `%pushd`, `%popd` and `%dhist`) and via direct `%cd`. The latter keeps a history of visited directories and allows you to go to any previously visited one.

Call system shell commands

Use Python to manipulate the results of system commands. The `!!` special syntax, and the `%sc` and `%sx` magic commands allow you to capture system output into Python variables.

Use Python variables when calling the shell

Expand python variables when calling the shell (either via `!` and `!!` or via aliases) by prepending a `$` in front of them. You can also expand complete python expressions. See [our shell section](#) for more details.

Use profiles

Use profiles to maintain different configurations (modules to load, function definitions, option settings) for particular tasks. You can then have customized versions of IPython for specific purposes. [This section](#) has more details.

Embed IPython in your programs

A few lines of code are enough to load a complete IPython inside your own programs, giving you the ability to work with your data interactively after automatic processing has been completed. See [here](#) for more.

Use the Python profiler

When dealing with performance issues, the `%run` command with a `-p` option allows you to run complete programs under the control of the Python profiler. The `%prun` command does a similar job for single Python expressions (like function calls).

Use IPython to present interactive demos

Use the `IPython.demo.Demo` class to load any Python script as an interactive demo. With a minimal amount of simple markup, you can control the execution of the script, stopping as needed. See [here](#) for more.

Run doctests

Run your doctests from within IPython for development and debugging. The special `%doctest_mode` command toggles a mode where the prompt, output and exceptions display matches as closely as possible that of the default Python interpreter. In addition, this mode allows you to directly paste in code that contains leading `>>>` prompts, even if they have extra leading whitespace (as is common in doctest files). This combined with the `%history -tn` call to see your translated history (with these extra prompts removed and no line numbers) allows for an easy doctest workflow, where you can go from doctest to interactive execution to pasting into valid Python code as needed.

4.1.2 Source code handling tips

IPython is a line-oriented program, without full control of the terminal. Therefore, it doesn't support true multiline editing. However, it has a number of useful tools to help you in dealing effectively with more complex editing.

The `%edit` command gives a reasonable approximation of multiline editing, by invoking your favorite editor on the spot. IPython will execute the code you type in there as if it were typed interactively. Type `%edit?` for the full details on the edit command.

If you have typed various commands during a session, which you'd like to reuse, IPython provides you with a number of tools. Start by using `%hist` to see your input history, so you can see the line numbers of all

input. Let us say that you'd like to reuse lines 10 through 20, plus lines 24 and 28. All the commands below can operate on these with the syntax:

```
%command 10-20 24 28
```

where the command given can be:

- `%macro <macroname>`: this stores the lines into a variable which, when called at the prompt, re-executes the input. Macros can be edited later using `%edit macroname`, and they can be stored persistently across sessions with `%store macroname` (the storage system is per-profile). The combination of quick macros, persistent storage and editing, allows you to easily refine quick-and-dirty interactive input into permanent utilities, always available both in IPython and as files for general reuse.
- `%edit`: this will open a text editor with those lines pre-loaded for further modification. It will then execute the resulting file's contents as if you had typed it at the prompt.
- `%save <filename>`: this saves the lines directly to a named file on disk.

While `%macro` saves input lines into memory for interactive re-execution, sometimes you'd like to save your input directly to a file. The `%save` magic does this: its input syntax is the same as `%macro`, but it saves your input directly to a Python file. Note that the `%logstart` command also saves input, but it logs all input to disk (though you can temporarily suspend it and reactivate it with `%logoff/%logon`); `%save` allows you to select which lines of input you need to save.

4.1.3 Lightweight 'version control'

When you call `%edit` with no arguments, IPython opens an empty editor with a temporary file, and it returns the contents of your editing session as a string variable. Thanks to IPython's output caching mechanism, this is automatically stored:

```
In [1]: %edit
```

```
IPython will make a temporary file named: /tmp/ipython_edit_yR-HCN.py
```

```
Editing... done. Executing edited code...
```

```
hello - this is a temporary file
```

```
Out[1]: "print 'hello - this is a temporary file'\n"
```

Now, if you call `%edit -p`, IPython tries to open an editor with the same data as the last time you used `%edit`. So if you haven't used `%edit` in the meantime, this same contents will reopen; however, it will be done in a new file. This means that if you make changes and you later want to find an old version, you can always retrieve it by using its output number, via `%edit _NN`, where NN is the number of the output prompt.

Continuing with the example above, this should illustrate this idea:

```
In [2]: edit -p
```

```
IPython will make a temporary file named: /tmp/ipython_edit_nA09Qk.py
```

```
Editing... done. Executing edited code...
```

```
hello - now I made some changes
```

```
Out[2]: "print 'hello - now I made some changes'\n"
```

```
In [3]: edit _1
```

```
IPython will make a temporary file named: /tmp/ipython_edit_gy6-zD.py
```

```
Editing... done. Executing edited code...
```

```
hello - this is a temporary file
```

```
IPython version control at work :)
```

```
Out[3]: "print 'hello - this is a temporary file'\nprint 'IPython version control at work"
```

This section was written after a contribution by Alexander Belchenko on the IPython user list.

4.1.4 Effective logging

A very useful suggestion sent in by Robert Kern follows:

I recently happened on a nifty way to keep tidy per-project log files. I made a profile for my project (which is called “parkfield”):

```
include ipythonrc

# cancel earlier logfile invocation:

logfile ''

execute import time

execute __cmd = '/Users/kern/research/logfiles/parkfield-%s.log rotate'

execute __IP.magic_logstart(__cmd % time.strftime('%Y-%m-%d'))
```

I also added a shell alias for convenience:

```
alias parkfield="ipython -pylab -profile parkfield"
```

Now I have a nice little directory with everything I ever type in, organized by project and date.

Contribute your own: If you have your own favorite tip on using IPython efficiently for a certain task (especially things which can’t be done in the normal Python interpreter), don’t hesitate to send it!

4.2 IPython reference

Warning: As of the 0.11 version of IPython, some of the features and APIs described in this section have been deprecated or are broken. Our plan is to continue to support these features, but they need to be updated to take advantage of recent API changes. Furthermore, this section of the documentation need to be updated to reflect all of these changes.

4.2.1 Command-line usage

You start IPython with the command:

```
$ ipython [options] files
```

If invoked with no options, it executes all the files listed in sequence and drops you into the interpreter while still acknowledging any options you may have set in your `ipythonrc` file. This behavior is different from standard Python, which when called as `python -i` will only execute one file and ignore your configuration setup.

Please note that some of the configuration options are not available at the command line, simply because they are not practical here. Look into your `ipythonrc` configuration file for details on those. This file typically installed in the `$HOME/.ipython` directory. For Windows users, `$HOME` resolves to `C:\Documents and Settings\YourUserName` in most instances. In the rest of this text, we will refer to this directory as `IPYTHON_DIR`.

Special Threading Options

Previously IPython had command line options for controlling GUI event loop integration (`-gthread`, `-qthread`, `-q4thread`, `-wthread`, `-pylab`). As of IPython version 0.11, these have been deprecated. Please see the new `%gui` magic command or [this section](#) for details on the new interface.

Regular Options

After the above threading options have been given, regular options can follow in any order. All options can be abbreviated to their shortest non-ambiguous form and are case-sensitive. One or two dashes can be used. Some options have an alternate short form, indicated after a `|`.

Most options can also be set from your `ipythonrc` configuration file. See the provided example for more details on what the options do. Options given at the command line override the values set in the `ipythonrc` file.

All options with a `[no]` prepended can be specified in negated form (`-nooption` instead of `-option`) to turn the feature off.

-help print a help message and exit.

`-pylab` Deprecated. See [Matplotlib support](#) for more details.

- autocall** <val> Make IPython automatically call any callable object even if you didn't type explicit parentheses. For example, 'str 43' becomes 'str(43)' automatically. The value can be '0' to disable the feature, '1' for smart autocall, where it is not applied if there are no more arguments on the line, and '2' for full autocall, where all callable objects are automatically called (even if no arguments are present). The default is '1'.
- [no]**autoindent** Turn automatic indentation on/off.
- [no]**automagic** make magic commands automatic (without needing their first character to be %). Type %magic at the IPython prompt for more information.
- [no]**autoedit_syntax** When a syntax error occurs after editing a file, automatically open the file to the trouble causing line for convenient fixing.
- [no]**banner** Print the initial information banner (default on).
 - c** <command> execute the given command string. This is similar to the -c option in the normal Python interpreter.
- cache_size, cs** <n> size of the output cache (maximum number of entries to hold in memory). The default is 1000, you can change it permanently in your config file. Setting it to 0 completely disables the caching system, and the minimum value accepted is 20 (if you provide a value less than 20, it is reset to 0 and a warning is issued) This limit is defined because otherwise you'll spend more time re-flushing a too small cache than working.
- classic, cl** Gives IPython a similar feel to the classic Python prompt.
- colors** <scheme> Color scheme for prompts and exception reporting. Currently implemented: NoColor, Linux and LightBG.
- [no]**color_info** IPython can display information about objects via a set of functions, and optionally can use colors for this, syntax highlighting source code and various other elements. However, because this information is passed through a pager (like 'less') and many pagers get confused with color codes, this option is off by default. You can test it and turn it on permanently in your ipythonrc file if it works for you. As a reference, the 'less' pager supplied with Mandrake 8.2 works ok, but that in RedHat 7.2 doesn't.

Test it and turn it on permanently if it works with your system. The magic function %color_info allows you to toggle this interactively for testing.
- [no]**debug** Show information about the loading process. Very useful to pin down problems with your configuration files or to get details about session restores.
- [no]**deep_reload**: IPython can use the deep_reload module which reloads changes in modules recursively (it replaces the reload() function, so you don't need to change anything to use it). deep_reload() forces a full reload of modules whose code may have changed, which the default reload() function does not.

When deep_reload is off, IPython will use the normal reload(), but deep_reload will still be available as dreload(). This feature is off by default [which means that you have both normal reload() and dreload()].
- editor** <name> Which editor to use with the %edit command. By default, IPython will honor your EDITOR environment variable (if not set, vi is the Unix default and notepad the Windows one). Since this editor is invoked on the fly by IPython and is meant for editing

small code snippets, you may want to use a small, lightweight editor here (in case your default EDITOR is something like Emacs).

-ipythondir <name> name of your IPython configuration directory IPYTHON_DIR. This can also be specified through the environment variable IPYTHON_DIR.

-log, l generate a log file of all input. The file is named ipython_log.py in your current directory (which prevents logs from multiple IPython sessions from trampling each other). You can use this to later restore a session by loading your logfile as a file to be executed with option -logplay (see below).

-logfile, lf <name> specify the name of your logfile.

-logplay, lp <name>

you can replay a previous log. For restoring a session as close as possible to the state you left it in, use this option (don't just run the logfile). With -logplay, IPython will try to reconstruct the previous working environment in full, not just execute the commands in the logfile.

When a session is restored, logging is automatically turned on again with the name of the logfile it was invoked with (it is read from the log header). So once you've turned logging on for a session, you can quit IPython and reload it as many times as you want and it will continue to log its history and restore from the beginning every time.

Caveats: there are limitations in this option. The history variables `_i*`, `_*` and `_dh` don't get restored properly. In the future we will try to implement full session saving by writing and retrieving a 'snapshot' of the memory state of IPython. But our first attempts failed because of inherent limitations of Python's Pickle module, so this may have to wait.

-[no]messages Print messages which IPython collects about its startup process (default on).

-[no]pdb Automatically call the pdb debugger after every uncaught exception. If you are used to debugging using pdb, this puts you automatically inside of it after any call (either in IPython or in code called by it) which triggers an exception which goes uncaught.

-pydb Makes IPython use the third party "pydb" package as debugger, instead of pdb. Requires that pydb is installed.

-[no]pprint ipython can optionally use the pprint (pretty printer) module for displaying results. pprint tends to give a nicer display of nested data structures. If you like it, you can turn it on permanently in your config file (default off).

-profile, p <name>

assume that your config file is ipythonrc-<name> or ipy_profile_<name>.py (looks in current dir first, then in IPYTHON_DIR). This is a quick way to keep and load multiple config files for different tasks, especially if you use the include option of config files. You can keep a basic IPYTHON_DIR/ipythonrc file and then have other 'profiles' which include this one and load extra things for particular tasks. For example:

1. \$HOME/.ipython/ipythonrc : load basic things you always want.

2. `$HOME/.ipython/ipythonrc-math` : load (1) and basic math-related modules.
3. `$HOME/.ipython/ipythonrc-numeric` : load (1) and Numeric and plotting modules.

Since it is possible to create an endless loop by having circular file inclusions, IPython will stop if it reaches 15 recursive inclusions.

-prompt_in1, pi1 <string>

Specify the string used for input prompts. Note that if you are using numbered prompts, the number is represented with a '#' in the string. Don't forget to quote strings with spaces embedded in them. Default: 'In [#]:'. The [prompts section](#) discusses in detail all the available escapes to customize your prompts.

-prompt_in2, pi2 <string> Similar to the previous option, but used for the continuation prompts. The special sequence 'D' is similar to '#', but with all digits replaced dots (so you can have your continuation prompt aligned with your input prompt). Default: '.D.: ' (note three spaces at the start for alignment with 'In [#]').

-prompt_out, po <string> String used for output prompts, also uses numbers like prompt_in1. Default: 'Out[#]:'

-quick start in bare bones mode (no config file loaded).

-rcfile <name> name of your IPython resource configuration file. Normally IPython loads `ipythonrc` (from current directory) or `IPYTHON_DIR/ipythonrc`.

If the loading of your config file fails, IPython starts with a bare bones configuration (no modules loaded at all).

-[no]readline use the readline library, which is needed to support name completion and command history, among other things. It is enabled by default, but may cause problems for users of X/Emacs in Python comint or shell buffers.

Note that X/Emacs 'eterm' buffers (opened with M-x term) support IPython's readline and syntax coloring fine, only 'emacs' (M-x shell and C-c !) buffers do not.

-screen_length, sl <n> number of lines of your screen. This is used to control printing of very long strings. Strings longer than this number of lines will be sent through a pager instead of directly printed.

The default value for this is 0, which means IPython will auto-detect your screen size every time it needs to print certain potentially long strings (this doesn't change the behavior of the 'print' keyword, it's only triggered internally). If for some reason this isn't working well (it needs curses support), specify it yourself. Otherwise don't change the default.

-separate_in, si <string>

separator before input prompts. Default: 'n'

-separate_out, so <string> separator before output prompts. Default: nothing.

-separate_out2, so2 separator after output prompts. Default: nothing. For these three options, use the value 0 to specify no separator.

- nosep** shorthand for ‘-SeparateIn 0 -SeparateOut 0 -SeparateOut2 0’. Simply removes all input/output separators.
- upgrade** allows you to upgrade your IPYTHON_DIR configuration when you install a new version of IPython. Since new versions may include new command line options or example files, this copies updated ipythonrc-type files. However, it backs up (with a .old extension) all files which it overwrites so that you can merge back any customizations you might have in your personal files. Note that you should probably use %upgrade instead, it’s a safer alternative.
- Version** print version information and exit.
- wxversion <string>** Deprecated.
- xmode <modename>**

Mode for exception reporting.

Valid modes: Plain, Context and Verbose.

- Plain: similar to python’s normal traceback printing.
- Context: prints 5 lines of context source code around each line in the traceback.
- Verbose: similar to Context, but additionally prints the variables currently visible where the exception happened (shortening their strings if too long). This can potentially be very slow, if you happen to have a huge data structure whose string representation is complex to compute. Your computer may appear to freeze for a while with cpu usage at 100%. If this occurs, you can cancel the traceback with Ctrl-C (maybe hitting it more than once).

4.2.2 Interactive use

Warning: IPython relies on the existence of a global variable called `_ip` which controls the shell itself. If you redefine `_ip` to anything, bizarre behavior will quickly occur.

Other than the above warning, IPython is meant to work as a drop-in replacement for the standard interactive interpreter. As such, any code which is valid python should execute normally under IPython (cases where this is not true should be reported as bugs). It does, however, offer many features which are not available at a standard python prompt. What follows is a list of these.

Caution for Windows users

Windows, unfortunately, uses the ‘`’` character as a path separator. This is a terrible choice, because ‘`’` also represents the escape character in most modern programming languages, including Python. For this reason, using ‘`/`’ character is recommended if you have problems with ‘`\`’. However, in Windows commands ‘`/`’ flags options, so you can not use it for the root directory. This means that paths beginning at the root must be typed in a contrived manner like: `%copy \opt/foo/bar.txt \tmp`

Magic command system

IPython will treat any line whose first character is a `%` as a special call to a ‘magic’ function. These allow you to control the behavior of IPython itself, plus a lot of system-type features. They are all prefixed with a `%` character, but parameters are given without parentheses or quotes.

Example: typing ‘`%cd mydir`’ (without the quotes) changes you working directory to ‘mydir’, if it exists.

If you have ‘automagic’ enabled (in your `ipythonrc` file, via the command line option `-automagic` or with the `%automagic` function), you don’t need to type in the `%` explicitly. IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type ‘`cd mydir`’ to go to directory ‘mydir’. The automagic system has the lowest possible precedence in name searches, so defining an identifier with the same name as an existing magic function will shadow it for automagic use. You can still access the shadowed magic function by explicitly using the `%` character at the beginning of the line.

An example (with automagic on) should clarify all this:

```
In [1]: cd ipython # %cd is called by automagic
/home/fperez/ipython

In [2]: cd=1 # now cd is just a variable

In [3]: cd .. # and doesn't work as a function anymore
-----
File "<console>", line 1
    cd ..
    ^
SyntaxError: invalid syntax

In [4]: %cd .. # but %cd always works
/home/fperez

In [5]: del cd # if you remove the cd variable

In [6]: cd ipython # automagic can work again
/home/fperez/ipython
```

You can define your own magic functions to extend the system. The following example defines a new magic command, `%impall`:

```
import IPython.ipapi

ip = IPython.ipapi.get()

def doimp(self, arg):
```

```
ip = self.api

ip.ex("import %s; reload(%s); from %s import *" % (
    arg, arg, arg)
)

ip.expose_magic('impall', doimp)
```

You can also define your own aliased names for magic functions. In your `ipythonrc` file, placing a line like:

```
execute __IP.magic_cl = __IP.magic_clear
```

will define `%cl` as a new name for `%clear`.

Type `%magic` for more information, including a list of all available magic functions at any time and their docstrings. You can also type `%magic_function_name?` (see sec. 6.4 <#sec:dyn-object-info> for information on the ‘?’ system) to get information about any particular magic function you are interested in.

The API documentation for the `IPython.Magic` module contains the full docstrings of all currently available magic commands.

Access to the standard Python help

As of Python 2.1, a help system is available with access to object docstrings and the Python manuals. Simply type ‘help’ (no quotes) to access it. You can also type `help(object)` to obtain information about a given object, and `help(‘keyword’)` for information on a keyword. As noted *here*, you need to properly configure your environment variable `PYTHONDOCS` for this feature to work correctly.

Dynamic object information

Typing `?word` or `word?` prints detailed information about an object. If certain strings in the object are too long (docstrings, code, etc.) they get snipped in the center for brevity. This system gives access variable types and values, full source code for any object (if available), function prototypes and other useful information.

Typing `??word` or `word??` gives access to the full information without snipping long strings. Long strings are sent to the screen through the less pager if longer than the screen and printed otherwise. On systems lacking the less command, IPython uses a very basic internal pager.

The following magic functions are particularly useful for gathering information about your working environment. You can get more details by typing `%magic` or querying them individually (use `%function_name?` with or without the `%`), this is just a summary:

- **%pdoc <object>**: Print (or run through a pager if too long) the docstring for an object. If the given object is a class, it will print both the class and the constructor docstrings.
- **%pdef <object>**: Print the definition header for any callable object. If the object is a class, print the constructor information.

- **%psource <object>**: Print (or run through a pager if too long) the source code for an object.
- **%pfile <object>**: Show the entire source file where an object was defined via a pager, opening it at the line where the object definition begins.
- **%who/%whos**: These functions give information about identifiers you have defined interactively (not things you loaded or defined in your configuration files). **%who** just prints a list of identifiers and **%whos** prints a table with some basic details about each identifier.

Note that the dynamic object information functions (**?/??**, **%pdoc**, **%pfile**, **%pdef**, **%psource**) give you access to documentation even on things which are not really defined as separate identifiers. Try for example typing `{ }.get?` or after doing `import os`, type `os.path.abspath??`.

Readline-based features

These features require the GNU readline library, so they won't work if your Python installation lacks readline support. We will first describe the default behavior IPython uses, and then how to change it to suit your preferences.

Command line completion

At any time, hitting TAB will complete any available python commands or variable names, and show you a list of the possible completions if there's no unambiguous one. It will also complete filenames in the current directory if no python names match what you've typed so far.

Search command history

IPython provides two ways for searching through previous input and thus reduce the need for repetitive typing:

1. Start typing, and then use Ctrl-p (previous,up) and Ctrl-n (next,down) to search through only the history items that match what you've typed so far. If you use Ctrl-p/Ctrl-n at a blank prompt, they just behave like normal arrow keys.
2. Hit Ctrl-r: opens a search prompt. Begin typing and the system searches your history for lines that contain what you've typed so far, completing as much as it can.

Persistent command history across sessions

IPython will save your input history when it leaves and reload it next time you restart it. By default, the history file is named `$IPYTHON_DIR/history`, but if you've loaded a named profile, `'-PROFILE_NAME'` is appended to the name. This allows you to keep separate histories related to various tasks: commands related to numerical work will not be clobbered by a system shell history, for example.

Autoindent

IPython can recognize lines ending in ‘:’ and indent the next line, while also un-indenting automatically after ‘raise’ or ‘return’.

This feature uses the readline library, so it will honor your ~/.inputrc configuration (or whatever file your INPUTRC variable points to). Adding the following lines to your .inputrc file can make indenting/unindenting more convenient (M-i indents, M-u unindents):

```
$if Python
"\M-i": "      "
"\M-u": "\d\d\d\d"
$endif
```

Note that there are 4 spaces between the quote marks after “M-i” above.

Warning: this feature is ON by default, but it can cause problems with the pasting of multi-line indented code (the pasted code gets re-indented on each line). A magic function %autoindent allows you to toggle it on/off at runtime. You can also disable it permanently on in your ipythonrc file (set autoindent 0).

Customizing readline behavior

All these features are based on the GNU readline library, which has an extremely customizable interface. Normally, readline is configured via a file which defines the behavior of the library; the details of the syntax for this can be found in the readline documentation available with your system or on the Internet. IPython doesn’t read this file (if it exists) directly, but it does support passing to readline valid options via a simple interface. In brief, you can customize readline by setting the following options in your ipythonrc configuration file (note that these options can not be specified at the command line):

- **readline_parse_and_bind**: this option can appear as many times as you want, each time defining a string to be executed via a readline.parse_and_bind() command. The syntax for valid commands of this kind can be found by reading the documentation for the GNU readline library, as these commands are of the kind which readline accepts in its configuration file.
- **readline_remove_delims**: a string of characters to be removed from the default word-delimiters list used by readline, so that completions may be performed on strings which contain them. Do not change the default value unless you know what you’re doing.
- **readline_omit_names**: when tab-completion is enabled, hitting <tab> after a ‘.’ in a name will complete all attributes of an object, including all the special methods whose names include double underscores (like __getitem__ or __class__). If you’d rather not see these names by default, you can set this option to 1. Note that even when this option is set, you can still see those names by explicitly typing a _ after the period and hitting <tab>: ‘name.<tab>’ will always complete attribute names starting with ‘_’.

This option is off by default so that new users see all attributes of any objects they are dealing with.

You will find the default values along with a corresponding detailed explanation in your ipythonrc file.

Session logging and restoring

You can log all input from a session either by starting IPython with the command line switches `-log` or `-logfile` (see [here](#)) or by activating the logging at any moment with the magic function `%logstart`.

Log files can later be reloaded with the `-logplay` option and IPython will attempt to ‘replay’ the log by executing all the lines in it, thus restoring the state of a previous session. This feature is not quite perfect, but can still be useful in many cases.

The log files can also be used as a way to have a permanent record of any code you wrote while experimenting. Log files are regular text files which you can later open in your favorite text editor to extract code or to ‘clean them up’ before using them to replay a session.

The `%logstart` function for activating logging in mid-session is used as follows:

```
%logstart [log_name [log_mode]]
```

If no name is given, it defaults to a file named ‘log’ in your `IPYTHON_DIR` 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):

- [over:] overwrite existing log_name.
- [backup:] rename (if exists) to log_name~ and start log_name.
- [append:] well, that says it.
- [rotate:] create rotating logs log_name.1~, log_name.2~, etc.

The `%logoff` and `%logon` functions allow you to temporarily stop and resume logging to a file which had previously been started with `%logstart`. They will fail (with an explanation) if you try to use them before logging has been started.

System shell access

Any input line beginning with a `!` character is passed verbatim (minus the `!`, of course) to the underlying operating system. For example, typing `!ls` will run ‘ls’ in the current directory.

Manual capture of command output

If the input line begins with two exclamation marks, `!!`, the command is executed but its output is captured and returned as a python list, split on newlines. Any output sent by the subprocess to standard error is printed separately, so that the resulting list only captures standard output. The `!!` syntax is a shorthand for the `%sx` magic command.

Finally, the `%sc` magic (short for ‘shell capture’) is similar to `%sx`, but allowing more fine-grained control of the capture details, and storing the result directly into a named variable. The direct use of `%sc` is now deprecated, and you should use the `var = !cmd` syntax instead.

IPython also allows you to expand the value of python variables when making system calls. Any python variable or expression which you prepend with `$` will get expanded before the system call is made:

```
In [1]: pyvar='Hello world'
In [2]: !echo "A python variable: $pyvar"
A python variable: Hello world
```

If you want the shell to actually see a literal `$`, you need to type it twice:

```
In [3]: !echo "A system variable: $$HOME"
A system variable: /home/fperez
```

You can pass arbitrary expressions, though you'll need to delimit them with `{ }` if there is ambiguity as to the extent of the expression:

```
In [5]: x=10
In [6]: y=20
In [13]: !echo $x+y
10+y
In [7]: !echo ${x+y}
30
```

Even object attributes can be expanded:

```
In [12]: !echo $sys.argv
[/home/fperez/usr/bin/ipython]
```

System command aliases

The `%alias` magic function and the `alias` option in the `ipythonrc` configuration file allow you to define magic functions which are in fact system shell commands. These aliases can have parameters.

`'%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).

You can also define aliases with parameters using `%s` specifiers (one per parameter). The following example defines the `%parts` function as an alias to the command `'echo first %s second %s'` where each `%s` will be replaced by a positional parameter to the call to `%parts`:

```
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'
```

If called with no parameters, `%alias` prints the table of currently defined aliases.

The `%rehash/rehashx` magics allow you to load your entire `$PATH` as ipython aliases. See their respective docstrings (or sec. 6.2 <#sec:magic> for further details).

Recursive reload

The `reload` function does a recursive reload of a module: changes made to the module since you imported will actually be available without having to exit.

Verbose and colored exception traceback printouts

IPython provides the option to see very detailed exception tracebacks, which can be especially useful when debugging large programs. You can run any Python file with the `%run` function to benefit from these detailed tracebacks. Furthermore, both normal and verbose tracebacks can be colored (if your terminal supports it) which makes them much easier to parse visually.

See the magic `xmode` and `colors` functions for details (just type `%magic`).

These features are basically a terminal version of Ka-Ping Yee's `cglib` module, now part of the standard Python library.

Input caching system

IPython offers numbered prompts (In/Out) with input and output caching (also referred to as 'input history'). All input is saved and can be retrieved as variables (besides the usual arrow key recall), in addition to the `%rep` magic command that brings a history entry up for editing on the next command line.

The following GLOBAL variables always exist (so don't overwrite them!): `_i`: stores previous input. `_ii`: next previous. `_iii`: next-next previous. `_ih`: a list of all input `_ih[n]` is the input from line `n` and this list is aliased to the global variable `In`. If you overwrite `In` with a variable of your own, you can remake the assignment to the internal list with a simple `In=_ih`.

Additionally, global variables named `_i<n>` are dynamically created (`<n>` being the prompt counter), such that `_i<n> == _ih[<n>] == In[<n>]`.

For example, what you typed at prompt 14 is available as `_i14`, `_ih[14]` and `In[14]`.

This allows you to easily cut and paste multi line interactive prompts by printing them out: they print like a clean string, without prompt characters. You can also manipulate them like regular variables (they are strings), modify or exec them (typing `'exec _i9'` will re-execute the contents of input prompt 9, `'exec In[9:14]+In[18]'` will re-execute lines 9 through 13 and line 18).

You can also re-execute multiple lines of input easily by using the magic `%macro` function (which automates the process and allows re-execution without having to type `'exec'` every time). The macro system also allows you to re-execute previous lines which include magic function calls (which require special processing). Type `%macro?` or see sec. 6.2 <#sec:magic> for more details on the macro system.

A history function `%hist` allows you to see any part of your input history by printing a range of the `_i` variables.

You can also search ('grep') through your history by typing `'%hist -g somestring'`. This also searches through the so called *shadow history*, which remembers all the commands (apart from multiline code blocks) you have ever entered. Handy for searching for svn/bzr URL's, IP addresses etc. You can bring shadow history entries listed by `'%hist -g'` up for editing (or re-execution by just pressing ENTER) with `%rep` command. Shadow history entries are not available as `_iNUMBER` variables, and they are identified by the

'0' prefix in `%hist -g` output. That is, history entry 12 is a normal history entry, but 0231 is a shadow history entry.

Shadow history was added because the readline history is inherently very unsafe - if you have multiple IPython sessions open, the last session to close will overwrite the history of previously closed session. Likewise, if a crash occurs, history is never saved, whereas shadow history entries are added after entering every command (so a command executed in another IPython session is immediately available in other IPython sessions that are open).

To conserve space, a command can exist in shadow history only once - it doesn't make sense to store a common line like `"cd .."` a thousand times. The idea is mainly to provide a reliable place where valuable, hard-to-remember commands can always be retrieved, as opposed to providing an exact sequence of commands you have entered in actual order.

Because shadow history has all the commands you have ever executed, time taken by `%hist -g` will increase over time. If it ever starts to take too long (or it ends up containing sensitive information like passwords), clear the shadow history by `%clear shadow_nuke`.

Time taken to add entries to shadow history should be negligible, but in any case, if you start noticing performance degradation after using IPython for a long time (or running a script that floods the shadow history!), you can 'compress' the shadow history by executing `%clear shadow_compress`. In practice, this should never be necessary in normal use.

Output caching system

For output that is returned from actions, a system similar to the input cache exists but using `_` instead of `_i`. Only actions that produce a result (NOT assignments, for example) are cached. If you are familiar with Mathematica, IPython's `_` variables behave exactly like Mathematica's `%` variables.

The following GLOBAL variables always exist (so don't overwrite them!):

- `[_]` (a single underscore) : stores previous output, like Python's default interpreter.
- `[_ _]` (two underscores): next previous.
- `[_ _ _]` (three underscores): next-next previous.

Additionally, global variables named `_<n>` are dynamically created (`<n>` being the prompt counter), such that the result of output `<n>` is always available as `_<n>` (don't use the angle brackets, just the number, e.g. `_21`).

These global variables are all stored in a global dictionary (not a list, since it only has entries for lines which returned a result) available under the names `_oh` and `Out` (similar to `_ih` and `In`). So the output from line 12 can be obtained as `_12`, `Out[12]` or `_oh[12]`. If you accidentally overwrite the `Out` variable you can recover it by typing `'Out=_oh'` at the prompt.

This system obviously can potentially put heavy memory demands on your system, since it prevents Python's garbage collector from removing any previously computed results. You can control how many results are kept in memory with the option (at the command line or in your `ipythonrc` file) `cache_size`. If you set it to 0, the whole system is completely disabled and the prompts revert to the classic `'>>>'` of normal Python.

Directory history

Your history of visited directories is kept in the global list `_dh`, and the magic `%cd` command can be used to go to any entry in that list. The `%dhist` command allows you to view this history. Do `cd -<TAB` to conveniently view the directory history.

Automatic parentheses and quotes

These features were adapted from Nathan Gray's LazyPython. They are meant to allow less typing for common situations.

Automatic parentheses

Callable objects (i.e. functions, methods, etc) can be invoked like this (notice the commas between the arguments):

```
>>> callable_ob arg1, arg2, arg3
```

and the input will be translated to this:

```
-> callable_ob(arg1, arg2, arg3)
```

You can force automatic parentheses by using `'` as the first character of a line. For example:

```
>>> /globals # becomes 'globals()'
```

Note that the `'` MUST be the first character on the line! This won't work:

```
>>> print /globals # syntax error
```

In most cases the automatic algorithm should work, so you should rarely need to explicitly invoke `/`. One notable exception is if you are trying to call a function with a list of tuples as arguments (the parenthesis will confuse IPython):

```
In [1]: zip (1,2,3), (4,5,6) # won't work
```

but this will work:

```
In [2]: /zip (1,2,3), (4,5,6)
--> zip ((1,2,3), (4,5,6))
Out[2]= [(1, 4), (2, 5), (3, 6)]
```

IPython tells you that it has altered your command line by displaying the new command line preceded by `->`. e.g.:

```
In [18]: callable list
----> callable (list)
```

Automatic quoting

You can force automatic quoting of a function's arguments by using `'` or `;` as the first character of a line. For example:

```
>>> ,my_function /home/me # becomes my_function("/home/me")
```

If you use `;` instead, the whole argument is quoted as a single string (while `'` splits on whitespace):

```
>>> ,my_function a b c # becomes my_function("a", "b", "c")
```

```
>>> ;my_function a b c # becomes my_function("a b c")
```

Note that the `'` or `;` MUST be the first character on the line! This won't work:

```
>>> x = ,my_function /home/me # syntax error
```

4.2.3 IPython as your default Python environment

Python honors the environment variable `PYTHONSTARTUP` and will execute at startup the file referenced by this variable. If you put at the end of this file the following two lines of code:

```
import IPython
IPython.Shell.IPShell().mainloop(sys_exit=1)
```

then IPython will be your working environment anytime you start Python. The `sys_exit=1` is needed to have IPython issue a call to `sys.exit()` when it finishes, otherwise you'll be back at the normal Python `>>>` prompt.

This is probably useful to developers who manage multiple Python versions and don't want to have correspondingly multiple IPython versions. Note that in this mode, there is no way to pass IPython any command-line options, as those are trapped first by Python itself.

4.2.4 Embedding IPython

It is possible to start an IPython instance inside your own Python programs. This allows you to evaluate dynamically the state of your code, operate with your variables, analyze them, etc. Note however that any changes you make to values while in the shell do not propagate back to the running code, so it is safe to modify your values because you won't break your code in bizarre ways by doing so.

This feature allows you to easily have a fully functional python environment for doing object introspection anywhere in your code with a simple function call. In some cases a simple print statement is enough, but if you need to do more detailed analysis of a code fragment this feature can be very valuable.

It can also be useful in scientific computing situations where it is common to need to do some automatic, computationally intensive part and then stop to look at data, plots, etc. Opening an IPython instance will give you full access to your data and functions, and you can resume program execution once you are done with the interactive part (perhaps to stop again later, as many times as needed).

The following code snippet is the bare minimum you need to include in your Python programs for this to work (detailed examples follow later):

```
from IPython.Shell import IPShellEmbed
```

```
ipshell = IPShellEmbed()
```

```
ipshell() # this call anywhere in your program will start IPython
```

You can run embedded instances even in code which is itself being run at the IPython interactive prompt with ‘%run <filename>’. Since it’s easy to get lost as to where you are (in your top-level IPython or in your embedded one), it’s a good idea in such cases to set the in/out prompts to something different for the embedded instances. The code examples below illustrate this.

You can also have multiple IPython instances in your program and open them separately, for example with different options for data presentation. If you close and open the same instance multiple times, its prompt counters simply continue from each execution to the next.

Please look at the docstrings in the Shell.py module for more details on the use of this system.

The following sample file illustrating how to use the embedding functionality is provided in the examples directory as example-embed.py. It should be fairly self-explanatory:

```
#!/usr/bin/env python
```

```
"""An example of how to embed an IPython shell into a running program.
```

```
Please see the documentation in the IPython.Shell module for more details.
```

```
The accompanying file example-embed-short.py has quick code fragments for  
embedding which you can cut and paste in your code once you understand how  
things work.
```

```
The code in this file is deliberately extra-verbose, meant for learning."""
```

```
# The basics to get you going:
```

```
# IPython sets the __IPYTHON__ variable so you can know if you have nested  
# copies running.
```

```
# Try running this code both at the command line and from inside IPython (with  
# %run example-embed.py)
```

```
try:
```

```
    __IPYTHON__
```

```
except NameError:
```

```
    nested = 0
```

```
    args = ['']
```

```
else:
```

```
    print "Running nested copies of IPython."
```

```
    print "The prompts for the nested copy have been modified"
```

```
    nested = 1
```

```
    # what the embedded instance will see as sys.argv:
```

```
    args = ['-pil', 'In <\\#>: ', '-pi2', '    .\\D.: ',  
           '-po', 'Out<\\#>: ', '-nosep']
```

```
# First import the embeddable shell class
```

```
from IPython.Shell import IPShellEmbed
```



```

# Now create an instance of the embeddable shell. The first argument is a
# string with options exactly as you would type them if you were starting
# IPython at the system command line. Any parameters you want to define for
# configuration can thus be specified here.
ipshell = IPShellEmbed(args,
                       banner = 'Dropping into IPython',
                       exit_msg = 'Leaving Interpreter, back to program.')

# Make a second instance, you can have as many as you want.
if nested:
    args[1] = 'In2<\\#>'
else:
    args = ['-pi1', 'In2<\\#>: ', '-pi2', ' .\\D.: ',
            '-po', 'Out<\\#>: ', '-nosep']
ipshell2 = IPShellEmbed(args, banner = 'Second IPython instance.')

print '\nHello. This is printed from the main controller program.\n'

# You can then call ipshell() anywhere you need it (with an optional
# message):
ipshell('***Called from top level. '
        'Hit Ctrl-D to exit interpreter and continue program.\n'
        'Note that if you use %kill_embedded, you can fully deactivate\n'
        'This embedded instance so it will never turn on again')

print '\nBack in caller program, moving along...\n'

#-----
# More details:

# IPShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
# messages) is available as <instance>.IP.BANNER in case you want it.

# IPShellEmbed instances print the following information everytime they
# start:

# - A global startup banner.

# - A call-specific header string, which you can use to indicate where in the
# execution flow the shell is starting.

# They also print an exit message every time they exit.

# Both the startup banner and the exit message default to None, and can be set
# either at the instance constructor or at any other time with the
# set_banner() and set_exit_msg() methods.

# The shell instance can be also put in 'dummy' mode globally or on a per-call
# basis. This gives you fine control for debugging without having to change
# code all over the place.

# The code below illustrates all this.

```

```
# This is how the global banner and exit_msg can be reset at any point
ipshell.set_banner('Entering interpreter - New Banner')
ipshell.set_exit_msg('Leaving interpreter - New exit_msg')

def foo(m):
    s = 'spam'
    ipshell('***In foo(). Try @whos, or print s or m:')
    print 'foo says m = ',m

def bar(n):
    s = 'eggs'
    ipshell('***In bar(). Try @whos, or print s or n:')
    print 'bar says n = ',n

# Some calls to the above functions which will trigger IPython:
print 'Main program calling foo("eggs")\n'
foo('eggs')

# The shell can be put in 'dummy' mode where calls to it silently return. This
# allows you, for example, to globally turn off debugging for a program with a
# single call.
ipshell.set_dummy_mode(1)
print '\nTrying to call IPython which is now "dummy":'
ipshell()
print 'Nothing happened...'
# The global 'dummy' mode can still be overridden for a single call
print '\nOverriding dummy mode manually:'
ipshell(dummy=0)

# Reactivate the IPython shell
ipshell.set_dummy_mode(0)

print 'You can even have multiple embedded instances:'
ipshell2()

print '\nMain program calling bar("spam")\n'
bar('spam')

print 'Main program finished. Bye!'

#***** End of file <example-embed.py> *****
```

Once you understand how the system functions, you can use the following code fragments in your programs which are ready for cut and paste:

```
"""Quick code snippets for embedding IPython into other programs.
```

```
See example-embed.py for full details, this file has the bare minimum code for
cut and paste use once you understand how to use the system."""
```

```
#-----
# This code loads IPython but modifies a few things if it detects it's running
```

```

# embedded in another IPython session (helps avoid confusion)

try:
    __IPYTHON__
except NameError:
    argv = ['']
    banner = exit_msg = ''
else:
    # Command-line options for IPython (a list like sys.argv)
    argv = ['-pil', 'In <\\#>:', '-pi2', '    .\\D.:', '-po', 'Out<\\#>:']
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'

# First import the embeddable shell class
from IPython.Shell import IPShellEmbed
# Now create the IPython shell instance. Put ipshell() anywhere in your code
# where you want it to open.
ipshell = IPShellEmbed(argv, banner=banner, exit_msg=exit_msg)

#-----
# This code will load an embeddable IPython shell always with no changes for
# nested embededings.

from IPython.Shell import IPShellEmbed
ipshell = IPShellEmbed()
# Now ipshell() will open IPython anywhere in the code.

#-----
# This code loads an embeddable shell only if NOT running inside
# IPython. Inside IPython, the embeddable shell variable ipshell is just a
# dummy function.

try:
    __IPYTHON__
except NameError:
    from IPython.Shell import IPShellEmbed
    ipshell = IPShellEmbed()
    # Now ipshell() will open IPython anywhere in the code
else:
    # Define a dummy ipshell() so the same code doesn't crash inside an
    # interactive IPython
    def ipshell(): pass

#***** End of file <example-embed-short.py> *****

```

4.2.5 Using the Python debugger (pdb)

Running entire programs via pdb

pdb, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of pdb,

regardless of whether you have wrapped it into a `'main()'` function or not. For this, simply type `'%run -d myscript'` at an IPython prompt. See the `%run` command's documentation (via `'%run?'` or in Sec. [magic](#) for more details, including how to control where pdb will stop execution first.

For more information on the use of the pdb debugger, read the included `pdb.doc` file (part of the standard Python distribution). On a stock Linux system it is located at `/usr/lib/python2.3/pdb.doc`, but the easiest way to read it is by using the `help()` function of the `pdb` module as follows (in an IPython prompt):

```
In [1]: import pdb In [2]: pdb.help()
```

This will load the `pdb.doc` document in a file viewer for you automatically.

Automatic invocation of pdb on exceptions

IPython, if started with the `-pdb` option (or if the option is set in your rc file) can call the Python pdb debugger every time your code triggers an uncaught exception. This feature can also be toggled at any time with the `%pdb` magic command. This can be extremely useful in order to find the origin of subtle bugs, because pdb opens up at the point in your code which triggered the exception, and while your program is at this point 'dead', all the data is still available and you can walk up and down the stack frame and understand the origin of the problem.

Furthermore, you can use these debugging facilities both with the embedded IPython mode and without IPython at all. For an embedded shell (see sec. [Embedding](#)), simply call the constructor with `'-pdb'` in the argument string and automatically pdb will be called if an uncaught exception is triggered by your code.

For stand-alone use of the feature in your programs which do not use IPython at all, put the following lines toward the top of your 'main' routine:

```
import sys
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
color_scheme='Linux', call_pdb=1)
```

The `mode` keyword can be either `'Verbose'` or `'Plain'`, giving either very detailed or normal tracebacks respectively. The `color_scheme` keyword can be one of `'NoColor'`, `'Linux'` (default) or `'LightBG'`. These are the same options which can be set in IPython with `-colors` and `-xmode`.

This will give any of your programs detailed, colored tracebacks with automatic invocation of pdb.

4.2.6 Extensions for syntax processing

This isn't for the faint of heart, because the potential for breaking things is quite high. But it can be a very powerful and useful feature. In a nutshell, you can redefine the way IPython processes the user input line to accept new, special extensions to the syntax without needing to change any of IPython's own code.

In the `IPython/extensions` directory you will find some examples supplied, which we will briefly describe now. These can be used 'as is' (and both provide very useful functionality), or you can use them as a starting point for writing your own extensions.

Pasting of code starting with '>>>' or '...'

In the python tutorial it is common to find code examples which have been taken from real python sessions. The problem with those is that all the lines begin with either '>>>' or '...', which makes it impossible to paste them all at once. One must instead do a line by line manual copying, carefully removing the leading extraneous characters.

This extension identifies those starting characters and removes them from the input automatically, so that one can paste multi-line examples directly into IPython, saving a lot of time. Please look at the file `InterpreterPasteInput.py` in the `IPython/extensions` directory for details on how this is done.

IPython comes with a special profile enabling this feature, called `tutorial`. Simply start IPython via `'ipython -p tutorial'` and the feature will be available. In a normal IPython session you can activate the feature by importing the corresponding module with: `In [1]: import IPython.extensions.InterpreterPasteInput`

The following is a 'screenshot' of how things work when this extension is on, copying an example from the standard tutorial:

```
IPython profile: tutorial
```

```
*** Pasting of code with ">>>" or "..." has been enabled.
```

```
In [1]: >>> def fib2(n): # return Fibonacci series up to n
...: ...     """Return a list containing the Fibonacci series up to
n."""
...: ...     result = []
...: ...     a, b = 0, 1
...: ...     while b < n:
...: ...         result.append(b)      # see below
...: ...         a, b = b, a+b
...: ...     return result
...:
```

```
In [2]: fib2(10)
```

```
Out[2]: [1, 1, 2, 3, 5, 8]
```

Note that as currently written, this extension does not recognize IPython's prompts for pasting. Those are more complicated, since the user can change them very easily, they involve numbers and can vary in length. One could however extract all the relevant information from the IPython instance and build an appropriate regular expression. This is left as an exercise for the reader.

Input of physical quantities with units

The module `PhysicalQInput` allows a simplified form of input for physical quantities with units. This file is meant to be used in conjunction with the `PhysicalQInteractive` module (in the same directory) and `Physics.PhysicalQuantities` from Konrad Hinsén's `ScientificPython` (<http://dirac.cnrs-orleans.fr/ScientificPython/>).

The `Physics.PhysicalQuantities` module defines `PhysicalQuantity` objects, but these must be declared as instances of a class. For example, to define `v` as a velocity of 3 m/s, normally you would write:

```
In [1]: v = PhysicalQuantity(3, 'm/s')
```

Using the `PhysicalQ_Input` extension this can be input instead as: `In [1]: v = 3 m/s` which is much more convenient for interactive use (even though it is blatantly invalid Python syntax).

The physics profile supplied with IPython (enabled via `'ipython -p physics'`) uses these extensions, which you can also activate with:

```
from math import * # math MUST be imported BEFORE PhysicalQInteractive from
IPython.extensions.PhysicalQInteractive import * import IPython.extensions.PhysicalQInput
```

4.2.7 GUI event loop support support

New in version 0.11: The `%gui` magic and `IPython.lib.inputhook`. IPython has excellent support for working interactively with Graphical User Interface (GUI) toolkits, such as `wxPython`, `PyQt4`, `PyGTK` and `Tk`. This is implemented using Python's builtin `PyOSInputHook` hook. This implementation is extremely robust compared to our previous threaded based version. The advantages of this are:

- GUIs can be enabled and disabled dynamically at runtime.
- The active GUI can be switched dynamically at runtime.
- In some cases, multiple GUIs can run simultaneously with no problems.
- There is a developer API in `IPython.lib.inputhook` for customizing all of these things.

For users, enabling GUI event loop integration is simple. You simply use the `%gui` magic as follows:

```
%gui [-a] [GUINAME]
```

With no arguments, `%gui` removes all GUI support. Valid `GUINAME` arguments are `wx`, `qt4`, `gtk` and `tk`. The `-a` option will create and return a running application object for the selected GUI toolkit.

Thus, to use `wxPython` interactively and create a running `wx.App` object, do:

```
%gui -a wx
```

For information on IPython's Matplotlib integration (and the `pylab` mode) see [this section](#).

For developers that want to use IPython's GUI event loop integration in the form of a library, these capabilities are exposed in library form in the `IPython.lib.inputhook`. Interested developers should see the module docstrings for more information, but there are a few points that should be mentioned here.

First, the `PyOSInputHook` approach only works in command line settings where `readline` is activated.

Second, when using the `PyOSInputHook` approach, a GUI application should *not* start its event loop. Instead all of this is handled by the `PyOSInputHook`. This means that applications that are meant to be used both in IPython and as standalone apps need to have special code to detect how the application is being run. We highly recommend using IPython's `appstart_()` functions for this. Here is a simple example that shows the recommended code that should be at the bottom of a `wxPython` using GUI application:

```
try:
    from IPython import appstart_wx
    appstart_wx(app)
```

```
except ImportError:
    app.MainLoop()
```

This pattern should be used instead of the simple `app.MainLoop()` code that a standalone wxPython application would have.

Third, unlike previous versions of IPython, we no longer “hijack” (replace them with no-ops) the event loops. This is done to allow applications that actually need to run the real event loops to do so. This is often needed to process pending events at critical points.

Finally, we also have a number of examples in our source directory `docs/examples/lib` that demonstrate these capabilities.

4.2.8 Plotting with matplotlib

Matplotlib provides high quality 2D and 3D plotting for Python. Matplotlib can produce plots on screen using a variety of GUI toolkits, including Tk, PyGTK, PyQt4 and wxPython. It also provides a number of commands useful for scientific computing, all with a syntax compatible with that of the popular Matlab program.

Many IPython users have come to rely on IPython’s `-pylab` mode which automates the integration of Matplotlib with IPython. We are still in the process of working with the Matplotlib developers to finalize the new pylab API, but for now you can use Matplotlib interactively using the following commands:

```
%gui -a wx
import matplotlib
matplotlib.use('wxagg')
from matplotlib import pylab
pylab.interactive(True)
```

All of this will soon be automated as Matplotlib begins to include new logic that uses our new GUI support.

4.2.9 Interactive demos with IPython

IPython ships with a basic system for running scripts interactively in sections, useful when presenting code to audiences. A few tags embedded in comments (so that the script remains valid Python code) divide a file into separate blocks, and the demo can be run one block at a time, with IPython printing (with syntax highlighting) the block before executing it, and returning to the interactive prompt after each block. The interactive namespace is updated after each block is run with the contents of the demo’s namespace.

This allows you to show a piece of code, run it and then execute interactively commands based on the variables just created. Once you want to continue, you simply execute the next block of the demo. The following listing shows the markup necessary for dividing a script into sections for execution as a demo:

```
"""A simple interactive demo to illustrate the use of IPython's Demo class.
```

```
Any python script can be run as a demo, but that does little more than showing
it on-screen, syntax-highlighted in one shot. If you add a little simple
markup, you can stop at specified intervals and return to the ipython prompt,
resuming execution later.
```

```
"""

print 'Hello, welcome to an interactive IPython demo.'
print 'Executing this block should require confirmation before proceeding,'
print 'unless auto_all has been set to true in the demo object'

# The mark below defines a block boundary, which is a point where IPython will
# stop execution and return to the interactive prompt.
# Note that in actual interactive execution,
# <demo> --- stop ---

x = 1
y = 2

# <demo> --- stop ---

# the mark below makes this block as silent
# <demo> silent

print 'This is a silent block, which gets executed but not printed.'

# <demo> --- stop ---
# <demo> auto
print 'This is an automatic block.'
print 'It is executed without asking for confirmation, but printed.'
z = x+y

print 'z=', x

# <demo> --- stop ---
# This is just another normal block.
print 'z is now:', z

print 'bye!'
```

In order to run a file as a demo, you must first make a Demo object out of it. If the file is named `myscript.py`, the following code will make a demo:

```
from IPython.demo import Demo

mydemo = Demo('myscript.py')
```

This creates the `mydemo` object, whose blocks you run one at a time by simply calling the object with no arguments. If you have `autocall` active in IPython (the default), all you need to do is type:

```
mydemo
```

and IPython will call it, executing each block. Demo objects can be restarted, you can move forward or back skipping blocks, re-execute the last block, etc. Simply use the Tab key on a demo object to see its methods, and call `?` on them to see their docstrings for more usage details. In addition, the demo module itself contains a comprehensive docstring, which you can access via:


```
from IPython import demo

demo?
```

Limitations: It is important to note that these demos are limited to fairly simple uses. In particular, you can not put division marks in indented code (loops, if statements, function definitions, etc.) Supporting something like this would basically require tracking the internal execution state of the Python interpreter, so only top-level divisions are allowed. If you want to be able to open an IPython instance at an arbitrary point in a program, you can use IPython’s embedding facilities, described in detail in Sec. 9

4.3 IPython as a system shell

Warning: As of the 0.11 version of IPython, some of the features and APIs described in this section have been deprecated or are broken. Our plan is to continue to support these features, but they need to be updated to take advantage of recent API changes. Furthermore, this section of the documentation need to be updated to reflect all of these changes.

4.3.1 Overview

The ‘sh’ profile optimizes IPython for system shell usage. Apart from certain job control functionality that is present in unix (ctrl+z does “suspend”), the sh profile should provide you with most of the functionality you use daily in system shell, and more. Invoke IPython in ‘sh’ profile by doing ‘ipython -p sh’, or (in win32) by launching the “pysh” shortcut in start menu.

If you want to use the features of sh profile as your defaults (which might be a good idea if you use other profiles a lot of the time but still want the convenience of sh profile), add `import ipy_profile_sh` to your `~/ipython/ipy_user_conf.py`.

The ‘sh’ profile is different from the default profile in that:

- Prompt shows the current directory
- Spacing between prompts and input is more compact (no padding with empty lines). The startup banner is more compact as well.
- System commands are directly available (in alias table) without requesting `%rehashx` - however, if you install new programs along your PATH, you might want to run `%rehashx` to update the persistent alias table
- Macros are stored in raw format by default. That is, instead of `‘_ip.system(“cat foo”)`, the macro will contain text `‘cat foo’`
- Autocall is in full mode
- Calling “up” does “cd ..”

The ‘sh’ profile is different from the now-obsolete (and unavailable) ‘pysh’ profile in that:

- `‘$$var = command’` and `‘$var = command’` syntax is not supported

- anymore. Use ‘var = !command’ instead (incidentally, this is
- available in all IPython profiles). Note that !!command *will*
- work.

4.3.2 Aliases

All of your \$PATH has been loaded as IPython aliases, so you should be able to type any normal system command and have it executed. See %alias? and %unalias? for details on the alias facilities. See also %rehashx? for details on the mechanism used to load \$PATH.

4.3.3 Directory management

Since each command passed by ipython to the underlying system is executed in a subshell which exits immediately, you can NOT use !cd to navigate the filesystem.

IPython provides its own builtin ‘%cd’ magic command to move in the filesystem (the % is not required with automatic on). It also maintains a list of visited directories (use %dhist to see it) and allows direct switching to any of them. Type ‘cd?’ for more details.

%pushd, %popd and %dirs are provided for directory stack handling.

4.3.4 Enabled extensions

Some extensions, listed below, are enabled as default in this profile.

envpersist

%env can be used to “remember” environment variable manipulations. Examples:

```
%env - Show all environment variables
%env VISUAL=jed - set VISUAL to jed
%env PATH+=;/foo - append ;foo to PATH
%env PATH+=;/bar - also append ;bar to PATH
%env PATH-=/wbin; - prepend /wbin; to PATH
%env -d VISUAL - forget VISUAL persistent val
%env -p - print all persistent env modifications
```

ipy_which

%which magic command. Like ‘which’ in unix, but knows about ipython aliases.

Example:

```
[C:/ipython]|14> %which st
st -> start .
[C:/ipython]|15> %which d
d -> dir /w /og /on
[C:/ipython]|16> %which cp
cp -> cp
    == c:\bin\cp.exe
c:\bin\cp.exe
```

ipy_app_completers

Custom tab completers for some apps like svn, hg, bzt, apt-get. Try ‘apt-get install <TAB>’ in debian/ubuntu.

ipy_rehashdir

Allows you to add system command aliases for commands that are not along your path. Let’s say that you just installed Putty and want to be able to invoke it without adding it to path, you can create the alias for it with rehashdir:

```
[~]|22> cd c:/opt/PuTTY/
[c:opt/PuTTY]|23> rehashdir .
    <23> ['pageant', 'plink', 'pscp', 'psftp', 'putty', 'puttygen', 'unins000']
```

Now, you can execute any of those commams directly:

```
[c:opt/PuTTY]|24> cd
[~]|25> putty
```

(the putty window opens).

If you want to store the alias so that it will always be available, do ‘%store putty’. If you want to %store all these aliases persistently, just do it in a for loop:

```
[~]|27> for a in _23:
|..>     %store $a
|..>
|..>
Alias stored: pageant (0, 'c:\\opt\\PuTTY\\pageant.exe')
Alias stored: plink (0, 'c:\\opt\\PuTTY\\plink.exe')
Alias stored: pscp (0, 'c:\\opt\\PuTTY\\pscp.exe')
Alias stored: psftp (0, 'c:\\opt\\PuTTY\\psftp.exe')
...
```

mglob

Provide the magic function %mglob, which makes it easier (than the ‘find’ command) to collect (possibly recursive) file lists. Examples:

```
[c:/ipython]|9> mglob *.py
[c:/ipython]|10> mglob *.py rec:*.txt
[c:/ipython]|19> workfiles = %mglob !.svn/ !.hg/ !*_Data/ !*.bak rec:.
```

Note that the first 2 calls will put the file list in result history (`_`, `_9`, `_10`), and the last one will assign it to ‘workfiles’.

4.3.5 Prompt customization

The sh profile uses the following prompt configurations:

```
o.prompt_in1= r'\C_LightBlue[\C_LightCyan\Y2\C_LightBlue]\C_Green|\#>'
o.prompt_in2= r'\C_Green|\C_LightGreen\D\C_Green>'
```

You can change the prompt configuration to your liking by editing `ipy_user_conf.py`.

4.3.6 String lists

String lists (`IPython.utils.text.SList`) are handy way to process output from system commands. They are produced by `var = !cmd syntax`.

First, we acquire the output of ‘ls -l’:

```
[Q:doc/examples]|2> lines = !ls -l
==
['total 23',
 '-rw-rw-rw- 1 ville None 1163 Sep 30 2006 example-demo.py',
 '-rw-rw-rw- 1 ville None 1927 Sep 30 2006 example-embed-short.py',
 '-rwxrwxrwx 1 ville None 4606 Sep 1 17:15 example-embed.py',
 '-rwxrwxrwx 1 ville None 1017 Sep 30 2006 example-gnuplot.py',
 '-rwxrwxrwx 1 ville None 339 Jun 11 18:01 extension.py',
 '-rwxrwxrwx 1 ville None 113 Dec 20 2006 seteditor.py',
 '-rwxrwxrwx 1 ville None 245 Dec 12 2006 seteditor.pyc']
```

Now, let’s take a look at the contents of ‘lines’ (the first number is the list element number):

```
[Q:doc/examples]|3> lines
<3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30 2006 example-demo.py
2: -rw-rw-rw- 1 ville None 1927 Sep 30 2006 example-embed-short.py
3: -rwxrwxrwx 1 ville None 4606 Sep 1 17:15 example-embed.py
4: -rwxrwxrwx 1 ville None 1017 Sep 30 2006 example-gnuplot.py
5: -rwxrwxrwx 1 ville None 339 Jun 11 18:01 extension.py
6: -rwxrwxrwx 1 ville None 113 Dec 20 2006 seteditor.py
7: -rwxrwxrwx 1 ville None 245 Dec 12 2006 seteditor.pyc
```

Now, let’s filter out the ‘embed’ lines:

```
[Q:doc/examples]|4> l2 = lines.grep('embed',prune=1)
[Q:doc/examples]|5> l2
<5> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30 2006 example-demo.py
2: -rwxrwxrwx 1 ville None 1017 Sep 30 2006 example-gnuplot.py
3: -rwxrwxrwx 1 ville None 339 Jun 11 18:01 extension.py
4: -rwxrwxrwx 1 ville None 113 Dec 20 2006 seteditor.py
5: -rwxrwxrwx 1 ville None 245 Dec 12 2006 seteditor.pyc
```

Now, we want strings having just file names and permissions:

```
[Q:doc/examples]|6> l2.fields(8,0)
<6> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total
1: example-demo.py -rw-rw-rw-
2: example-gnuplot.py -rwxrwxrwx
3: extension.py -rwxrwxrwx
4: seteditor.py -rwxrwxrwx
5: seteditor.pyc -rwxrwxrwx
```

Note how the line with ‘total’ does not raise `IndexError`.

If you want to split these (yielding lists), call `fields()` without arguments:

```
[Q:doc/examples]|7> _.fields()
<7>

[['total'],
 ['example-demo.py', '-rw-rw-rw-'],
 ['example-gnuplot.py', '-rwxrwxrwx'],
 ['extension.py', '-rwxrwxrwx'],
 ['seteditor.py', '-rwxrwxrwx'],
 ['seteditor.pyc', '-rwxrwxrwx']]
```

If you want to pass these separated with spaces to a command (typical for lists of files), use the `.s` property:

```
[Q:doc/examples]|13> files = l2.fields(8).s
[Q:doc/examples]|14> files
<14> 'example-demo.py example-gnuplot.py extension.py seteditor.py seteditor.pyc'
[Q:doc/examples]|15> ls $files
example-demo.py example-gnuplot.py extension.py seteditor.py seteditor.pyc
```

SLists are inherited from normal python lists, so every list method is available:

```
[Q:doc/examples]|21> lines.append('hey')
```

4.3.7 Real world example: remove all files outside version control

First, capture output of “hg status”:

```
[Q:/ipython]|28> out = !hg status
==
['M IPython\\extensions\\ipy_kitcfg.py',
'M IPython\\extensions\\ipy_rehashdir.py',
...
'? build\\lib\\IPython\\Debugger.py',
'? build\\lib\\IPython\\extensions\\InterpreterExec.py',
'? build\\lib\\IPython\\extensions\\InterpreterPasteInput.py',
...]
```

(lines starting with ? are not under version control).

```
[Q:/ipython]|35> junk = out.grep(r'^\?').fields(1)
[Q:/ipython]|36> junk
<36> SList (.p, .n, .l, .s, .grep(), .fields() availab
...
10: build\bdist.win32\winexe\temp\_ctypes.py
11: build\bdist.win32\winexe\temp\_hashlib.py
12: build\bdist.win32\winexe\temp\_socket.py
```

Now we can just remove these files by doing ‘rm \$junk.s’.

4.3.8 The .s, .n, .p properties

The ‘.s’ property returns one string where lines are separated by single space (for convenient passing to system commands). The ‘.n’ property return one string where the lines are separated by ‘n’ (i.e. the original output of the function). If the items in string list are file names, ‘.p’ can be used to get a list of “path” objects for convenient file manipulation.

4.4 IPython as a QtGUI widget

We now have a version of IPython, using the new two-process *ZeroMQ Kernel*, running in a *PyQt* GUI.

4.4.1 Overview

The Qt frontend has hand-coded emacs-style bindings for text navigation. This is not yet configurable.

See Also:

The original IPython-Qt project description.

4.4.2 %loadpy

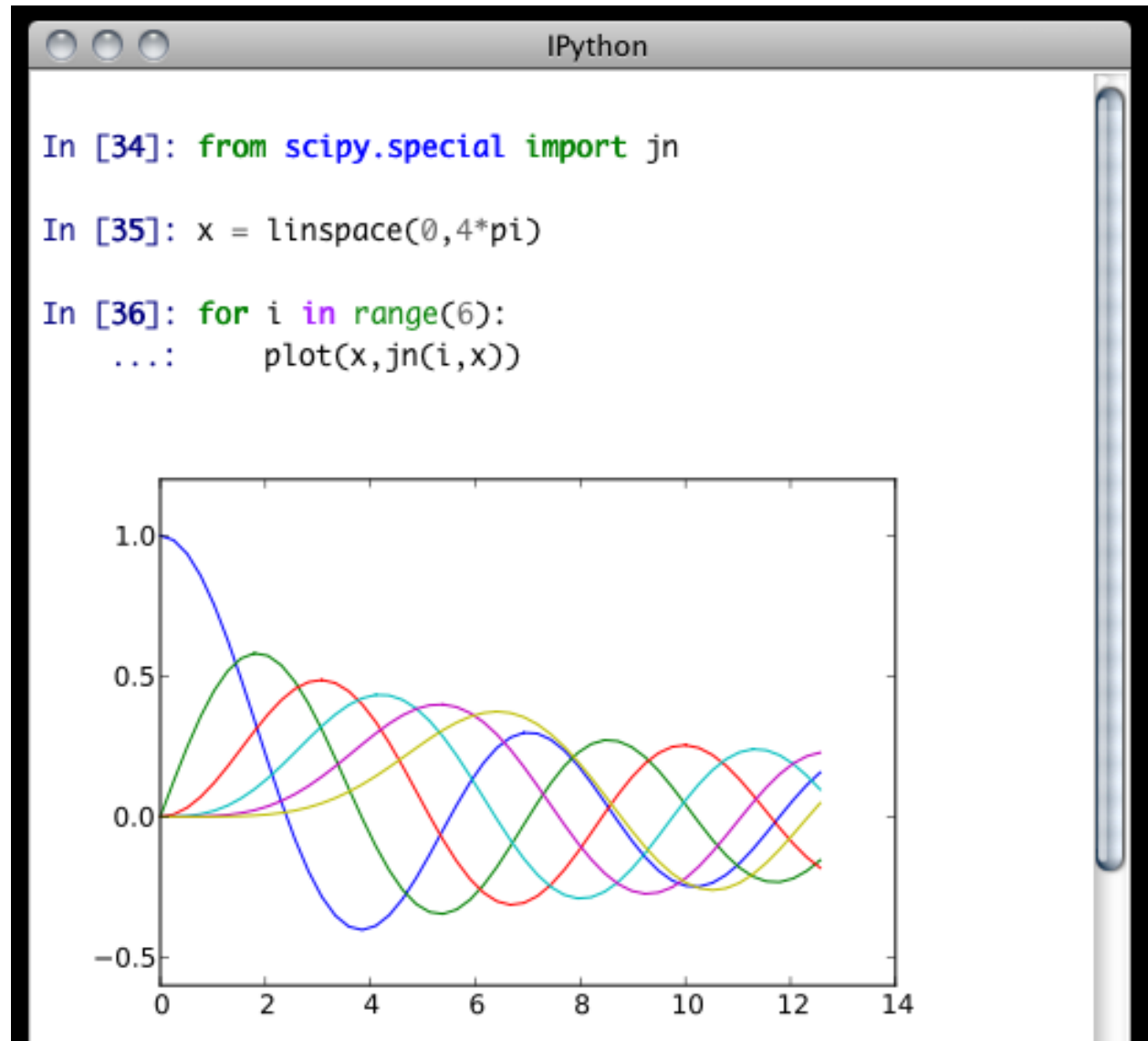
The %loadpy magic has been added, just for the GUI frontend. It takes any python script (must end in ‘.py’), and pastes its contents as your next input, so you can edit it before executing. The script may be on your machine, but you can also specify a url, and it will download the script from the web. This is particularly useful for playing with examples from documentation, such as matplotlib.

```
In [6]: %loadpy
http://matplotlib.sourceforge.net/plot_directive/mpl_examples/mplot3d/contour3d_demo.py

In [7]: from mpl_toolkits.mplot3d import axes3d
...: import matplotlib.pyplot as plt
...:
...: fig = plt.figure()
...: ax = fig.add_subplot(111, projection='3d')
...: X, Y, Z = axes3d.get_test_data(0.05)
...: cset = ax.contour(X, Y, Z)
...: ax.clabel(cset, fontsize=9, inline=1)
...:
...: plt.show()
```

4.4.3 Pylab

One of the most exciting features of the new console is embedded matplotlib figures. You can use any standard matplotlib GUI backend (Except native MacOSX) to draw the figures, and since there is now a two-process model, there is no longer a conflict between user input and the drawing eventloop.



`pastefig()`

An additional function, `pastefig()`, will be added to the global namespace if you specify the `--pylab` argument. This takes the active figures in matplotlib, and embeds them in your document. This is especially useful for [saving](#) your work.

`--pylab inline`

If you want to have all of your figures embedded in your session, instead of calling `pastefig()`, you can specify `--pylab inline`, and each time you make a plot, it will show up in your document, as if you had called `pastefig()`.

4.4.4 Saving and Printing

IPythonQt has the ability to save your current session, as either HTML or XHTML. If you have been using `pastefig` or `inline` `pylab`, your figures will be PNG in HTML, or inlined as SVG in XHTML. PNG images have the option to be either in an external folder, as in many browsers' "Webpage, Complete" option, or inlined as well, for a larger, but more portable file.

The widget also exposes the ability to print directly, via the default print shortcut or context menu.

Note: Saving is only available to richtext Qt widgets, so make sure you start `ipqt` with the `--rich` flag, or with `--pylab`, which always uses a richtext widget.

See these examples of `png/html` and `svg/xhtml` output. Note that syntax highlighting does not survive export. This is a known issue, and is being investigated.

4.4.5 Colors and Highlighting

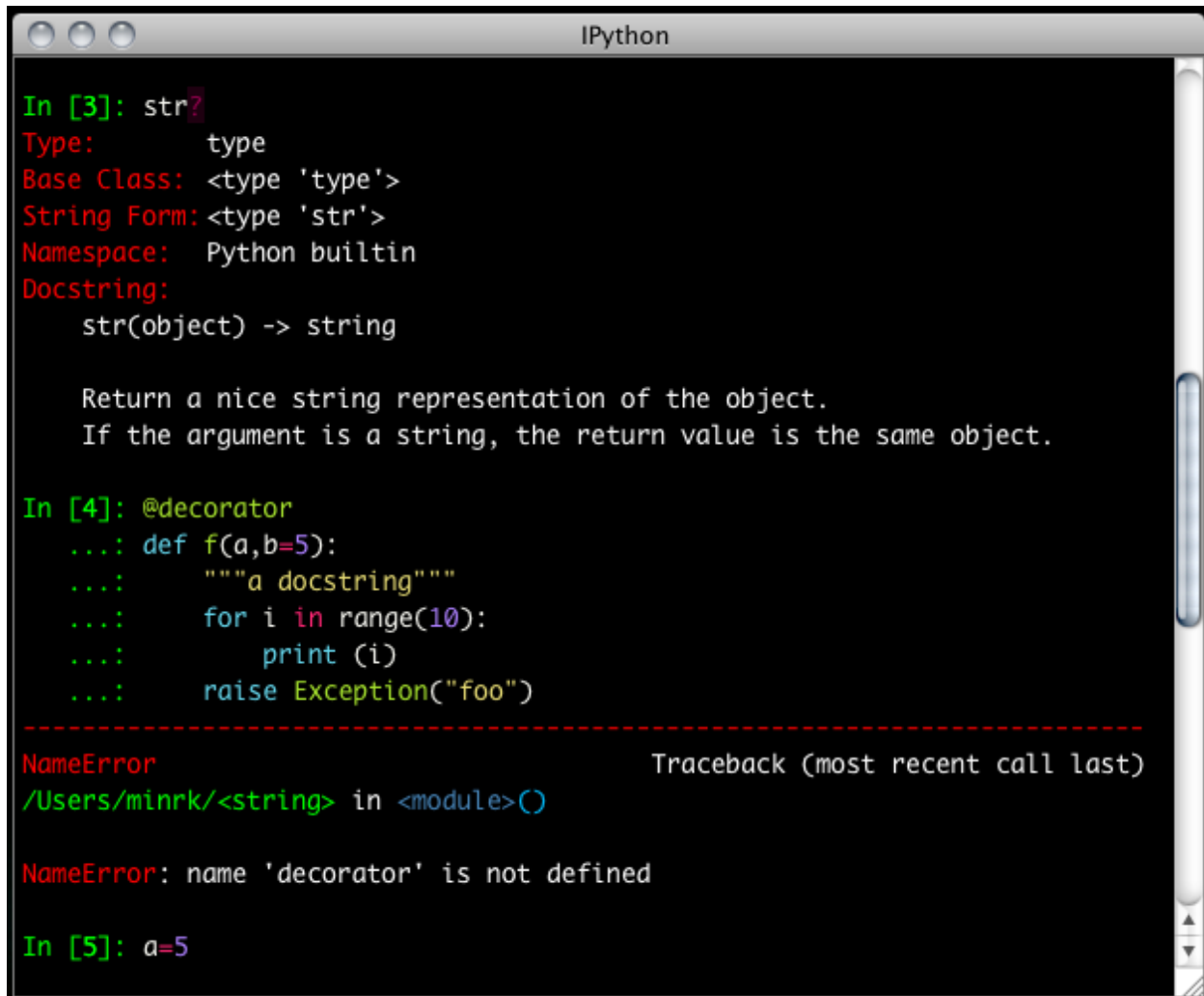
Terminal IPython has always had some coloring, but never syntax highlighting. There are a few simple color choices, specified by the `--colors` flag or `%colors` magic:

- `LightBG` for light backgrounds
- `Linux` for dark backgrounds
- `NoColor` for a simple colorless terminal

The Qt widget has full support for the `--colors` flag, but adds new, more intuitive aliases for the colors (the old names still work): `dark=Linux`, `light=LightBG`, `bw=NoColor`.

The Qt widget, however, has full syntax highlighting as you type, handled by the `pygments` library. The `--style` argument exposes access to any style by name that can be found by `pygments`, and there are several already installed. The `--colors` argument, if unspecified, will be guessed based on the chosen style. Similarly, there are default styles associated with each `--colors` option.

Screenshot of `ipython-qtconsole --colors dark`, which uses the 'monokai' theme by default:



```
IPython

In [3]: str?
Type:      type
Base Class: <type 'type'>
String Form: <type 'str'>
Namespace: Python builtin
Docstring:
    str(object) -> string

    Return a nice string representation of the object.
    If the argument is a string, the return value is the same object.

In [4]: @decorator
...: def f(a,b=5):
...:     """a docstring"""
...:     for i in range(10):
...:         print (i)
...:     raise Exception("foo")
-----
NameError                                Traceback (most recent call last)
/Users/minrk/<string> in <module>()

NameError: name 'decorator' is not defined

In [5]: a=5
```

Note: Calling `ipython-qtconsole -h` will show all the style names that pygments can find on your system.

You can also pass the filename of a custom CSS stylesheet, if you want to do your own coloring, via the `--stylesheet` argument. The default LightBG stylesheet:

```
QPlainTextEdit, QTextEdit { background-color: white;
    color: black ;
    selection-background-color: #ccc}
.error { color: red; }
.in-prompt { color: navy; }
.in-prompt-number { font-weight: bold; }
.out-prompt { color: darkred; }
.out-prompt-number { font-weight: bold; }
```

4.4.6 Process Management

With the two-process ZMQ model, the frontend does not block input during execution. This means that actions can be taken by the frontend while the Kernel is executing, or even after it crashes. The most basic such command is via ‘Ctrl-.’, which restarts the kernel. This can be done in the middle of a blocking execution. The frontend can also know, via a heartbeat mechanism, that the kernel has died. This means that the frontend can safely restart the kernel.

Multiple Consoles

Since the Kernel listens on the network, multiple frontends can connect to it. These do not have to all be qt frontends - any IPython frontend can connect and run code. When you start `ipython-qtconsole`, there will be an output line, like:

```
To connect another client to this kernel, use:
-e --xreq 62109 --sub 62110 --rep 62111 --hb 62112
```

Other frontends can connect to your kernel, and share in the execution. This is great for collaboration. The `-e` flag is for ‘external’. Starting other consoles with that flag will not try to start their own, but rather connect to yours. Ultimately, you will not have to specify each port individually, but for now this copy-paste method is best.

By default (for security reasons), the kernel only listens on localhost, so you can only connect multiple frontends to the kernel from your local machine. You can specify to listen on an external interface by specifying the `--ip` argument:

```
$> ipython-qtconsole --ip 192.168.1.123
```

If you specify the ip as 0.0.0.0, that refers to all interfaces, so any computer that can see yours can connect to the kernel.

Warning: Since the ZMQ code currently has no security, listening on an external-facing IP is dangerous. You are giving any computer that can see you on the network the ability to issue arbitrary shell commands as you on your machine. Be very careful with this.

Stopping Kernels and Consoles

Since there can be many consoles per kernel, the shutdown mechanism and dialog are probably more complicated than you are used to. Since you don’t always want to shutdown a kernel when you close a window, you are given the option to just close the console window or also close the Kernel and *all other windows*. Note that this only refers to all other *local* windows, as remote Consoles are not allowed to shutdown the kernel, and shutdowns do not close Remote consoles (to allow for saving, etc.).

Rules:

- Restarting the kernel automatically clears all *local* Consoles, and prompts remote Consoles about the reset.
- Shutdown closes all *local* Consoles, and notifies remotes that the Kernel has been shutdown.

- Remote Consoles may not restart or shutdown the kernel.

4.4.7 Regressions

There are some features, where the qt console lags behind the Terminal frontend. We hope to have these fixed by 0.11 release.

- Configuration: The Qt frontend and ZMQ kernel are not yet hooked up to the IPython configuration system
- History Persistence: Currently the history of a GUI session does not persist between sessions.
- !cmd input: Due to our use of pexpect, we cannot pass input to subprocesses launched using the ‘!’ escape. (this will not be fixed).

USING IPYTHON FOR PARALLEL COMPUTING

5.1 Overview and getting started

5.1.1 Introduction

This section gives an overview of IPython's sophisticated and powerful architecture for parallel and distributed computing. This architecture abstracts out parallelism in a very general way, which enables IPython to support many different styles of parallelism including:

- Single program, multiple data (SPMD) parallelism.
- Multiple program, multiple data (MPMD) parallelism.
- Message passing using MPI.
- Task farming.
- Data parallel.
- Combinations of these approaches.
- Custom user defined approaches.

Most importantly, IPython enables all types of parallel applications to be developed, executed, debugged and monitored *interactively*. Hence, the \mathbb{I} in IPython. The following are some example usage cases for IPython:

- Quickly parallelize algorithms that are embarrassingly parallel using a number of simple approaches. Many simple things can be parallelized interactively in one or two lines of code.
- Steer traditional MPI applications on a supercomputer from an IPython session on your laptop.
- Analyze and visualize large datasets (that could be remote and/or distributed) interactively using IPython and tools like matplotlib/TVTK.
- Develop, test and debug new parallel algorithms (that may use MPI) interactively.
- Tie together multiple MPI jobs running on different systems into one giant distributed and parallel system.

- Start a parallel job on your cluster and then have a remote collaborator connect to it and pull back data into their local IPython session for plotting and analysis.
- Run a set of tasks on a set of CPUs using dynamic load balancing.

5.1.2 Architecture overview

The IPython architecture consists of three components:

- The IPython engine.
- The IPython controller.
- Various controller clients.

These components live in the `IPython.kernel` package and are installed with IPython. They do, however, have additional dependencies that must be installed. For more information, see our [installation documentation](#).

IPython engine

The IPython engine is a Python instance that takes Python commands over a network connection. Eventually, the IPython engine will be a full IPython interpreter, but for now, it is a regular Python interpreter. The engine can also handle incoming and outgoing Python objects sent over a network connection. When multiple engines are started, parallel and distributed computing becomes possible. An important feature of an IPython engine is that it blocks while user code is being executed. Read on for how the IPython controller solves this problem to expose a clean asynchronous API to the user.

IPython controller

The IPython controller provides an interface for working with a set of engines. At a general level, the controller is a process to which IPython engines can connect. For each connected engine, the controller manages a queue. All actions that can be performed on the engine go through this queue. While the engines themselves block when user code is run, the controller hides that from the user to provide a fully asynchronous interface to a set of engines.

Note: Because the controller listens on a network port for engines to connect to it, it must be started *before* any engines are started.

The controller also provides a single point of contact for users who wish to utilize the engines connected to the controller. There are different ways of working with a controller. In IPython these ways correspond to different interfaces that the controller is adapted to. Currently we have two default interfaces to the controller:

- The MultiEngine interface, which provides the simplest possible way of working with engines interactively.
- The Task interface, which presents the engines as a load balanced task farming system.

Advanced users can easily add new custom interfaces to enable other styles of parallelism.

Note: A single controller and set of engines can be accessed through multiple interfaces simultaneously. This opens the door for lots of interesting things.

Controller clients

For each controller interface, there is a corresponding client. These clients allow users to interact with a set of engines through the interface. Here are the two default clients:

- The `MultiEngineClient` class.
- The `TaskClient` class.

Security

By default (as long as *pyOpenSSL* is installed) all network connections between the controller and engines and the controller and clients are secure. What does this mean? First of all, all of the connections will be encrypted using SSL. Second, the connections are authenticated. We handle authentication in a capability based security model [Capability]. In this model, a “capability (known in some systems as a key) is a communicable, unforgeable token of authority”. Put simply, a capability is like a key to your house. If you have the key to your house, you can get in. If not, you can’t.

In our architecture, the controller is the only process that listens on network ports, and is thus responsible to creating these keys. In IPython, these keys are known as Foolscape URLs, or FURLs, because of the underlying network protocol we are using. As a user, you don’t need to know anything about the details of these FURLs, other than that when the controller starts, it saves a set of FURLs to files named `something.furl`. The default location of these files is the `~/ipython/security` directory.

To connect and authenticate to the controller an engine or client simply needs to present an appropriate FURL (that was originally created by the controller) to the controller. Thus, the FURL files need to be copied to a location where the clients and engines can find them. Typically, this is the `~/ipython/security` directory on the host where the client/engine is running (which could be a different host than the controller). Once the FURL files are copied over, everything should work fine.

Currently, there are three FURL files that the controller creates:

ipcontroller-engine.furl This FURL file is the key that gives an engine the ability to connect to a controller.

ipcontroller-tc.furl This FURL file is the key that a `TaskClient` must use to connect to the task interface of a controller.

ipcontroller-mec.furl This FURL file is the key that a `MultiEngineClient` must use to connect to the multiengine interface of a controller.

More details of how these FURL files are used are given below.

A detailed description of the security model and its implementation in IPython can be found [here](#).

5.1.3 Getting Started

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. Initially, it is best to simply start a controller and engines on a single host using the **ipcluster** command. To start a controller and 4 engines on your localhost, just do:

```
$ ipcluster local -n 4
```

More details about starting the IPython controller and engines can be found [here](#)

Once you have started the IPython controller and one or more engines, you are ready to use the engines to do something useful. To make sure everything is working correctly, try the following commands:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()

In [4]: mec.get_ids()
Out[4]: [0, 1, 2, 3]

In [5]: mec.execute('print "Hello World"')
Out[5]:
<Results List>
[0] In [1]: print "Hello World"
[0] Out[1]: Hello World

[1] In [1]: print "Hello World"
[1] Out[1]: Hello World

[2] In [1]: print "Hello World"
[2] Out[1]: Hello World

[3] In [1]: print "Hello World"
[3] Out[1]: Hello World
```

Remember, a client also needs to present a FURL file to the controller. How does this happen? When a multiengine client is created with no arguments, the client tries to find the corresponding FURL file in the local `~/ipython/security` directory. If it finds it, you are set. If you have put the FURL file in a different location or it has a different name, create the client like this:

```
mec = client.MultiEngineClient('/path/to/my/ipcontroller-mec.furl')
```

Same thing hold true of creating a task client:

```
tc = client.TaskClient('/path/to/my/ipcontroller-tc.furl')
```

You are now ready to learn more about the *MultiEngine* and *Task* interfaces to the controller.

Note: Don't forget that the engine, multiengine client and task client all have *different* furl files. You must move *each* of these around to an appropriate location so that the engines and clients can use them to connect to the controller.

5.2 Starting the IPython controller and engines

To use IPython for parallel computing, you need to start one instance of the controller and one or more instances of the engine. The controller and each engine can run on different machines or on the same machine. Because of this, there are many different possibilities.

Broadly speaking, there are two ways of going about starting a controller and engines:

- In an automated manner using the **ipcluster** command.
- In a more manual way using the **ipcontroller** and **ipengine** commands.

This document describes both of these methods. We recommend that new users start with the **ipcluster** command as it simplifies many common usage cases.

5.2.1 General considerations

Before delving into the details about how you can start a controller and engines using the various methods, we outline some of the general issues that come up when starting the controller and engines. These things come up no matter which method you use to start your IPython cluster.

Let's say that you want to start the controller on `host0` and engines on hosts `host1-hostn`. The following steps are then required:

1. Start the controller on `host0` by running **ipcontroller** on `host0`.
2. Move the FURL file (`ipcontroller-engine.furl`) created by the controller from `host0` to hosts `host1-hostn`.
3. Start the engines on hosts `host1-hostn` by running **ipengine**. This command has to be told where the FURL file (`ipcontroller-engine.furl`) is located.

At this point, the controller and engines will be connected. By default, the FURL files created by the controller are put into the `~/ipython/security` directory. If the engines share a filesystem with the controller, step 2 can be skipped as the engines will automatically look at that location.

The final step required required to actually use the running controller from a client is to move the FURL files `ipcontroller-mec.furl` and `ipcontroller-tc.furl` from `host0` to the host where the clients will be run. If these file are put into the `~/ipython/security` directory of the client's host, they will be found automatically. Otherwise, the full path to them has to be passed to the client's constructor.

5.2.2 Using ipcluster

The **ipcluster** command provides a simple way of starting a controller and engines in the following situations:

1. When the controller and engines are all run on `localhost`. This is useful for testing or running on a multicore computer.
2. When engines are started using the **mpirun** command that comes with most MPI [\[MPI\]](#) implementations

3. When engines are started using the PBS [\[PBS\]](#) batch system.
4. When the controller is started on localhost and the engines are started on remote nodes using **ssh**.

Note: It is also possible for advanced users to add support to **ipcluster** for starting controllers and engines using other methods (like Sun's Grid Engine for example).

Note: Currently **ipcluster** requires that the `~/.ipython/security` directory live on a shared filesystem that is seen by both the controller and engines. If you don't have a shared file system you will need to use **ipcontroller** and **ipengine** directly. This constraint can be relaxed if you are using the **ssh** method to start the cluster.

Underneath the hood, **ipcluster** just uses **ipcontroller** and **ipengine** to perform the steps described above.

Using ipcluster in local mode

To start one controller and 4 engines on localhost, just do:

```
$ ipcluster local -n 4
```

To see other command line options for the local mode, do:

```
$ ipcluster local -h
```

Using ipcluster in mpiexec/mpirun mode

The mpiexec/mpirun mode is useful if you:

1. Have MPI installed.
2. Your systems are configured to use the **mpiexec** or **mpirun** commands to start MPI processes.

Note: The preferred command to use is **mpiexec**. However, we also support **mpirun** for backwards compatibility. The underlying logic used is exactly the same, the only difference being the name of the command line program that is called.

If these are satisfied, you can start an IPython cluster using:

```
$ ipcluster mpiexec -n 4
```

This does the following:

1. Starts the IPython controller on current host.
2. Uses **mpiexec** to start 4 engines.

On newer MPI implementations (such as OpenMPI), this will work even if you don't make any calls to MPI or call `MPI_Init()`. However, older MPI implementations actually require each process to call

`MPI_Init()` upon starting. The easiest way of having this done is to install the `mpi4py` [mpi4py] package and then call `ipcluster` with the `--mpi` option:

```
$ ipcluster mpiexec -n 4 --mpi=mpi4py
```

Unfortunately, even this won't work for some MPI implementations. If you are having problems with this, you will likely have to use a custom Python executable that itself calls `MPI_Init()` at the appropriate time. Fortunately, `mpi4py` comes with such a custom Python executable that is easy to install and use. However, this custom Python executable approach will not work with **ipcluster** currently.

Additional command line options for this mode can be found by doing:

```
$ ipcluster mpiexec -h
```

More details on using MPI with IPython can be found [here](#).

Using ipcluster in PBS mode

The PBS mode uses the Portable Batch System [PBS] to start the engines. To use this mode, you first need to create a PBS script template that will be used to start the engines. Here is a sample PBS script template:

```
#PBS -N ipython
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l nodes=${n/4}:ppn=4
#PBS -q parallel

cd $$PBS_O_WORKDIR
export PATH=$$HOME/usr/local/bin
export PYTHONPATH=$$HOME/usr/local/lib/python2.4/site-packages
/usr/local/bin/mpiexec -n ${n} ipengine --logfile=$$PBS_O_WORKDIR/ipengine
```

There are a few important points about this template:

1. This template will be rendered at runtime using IPython's `Itpl` template engine.
2. Instead of putting in the actual number of engines, use the notation `${n}` to indicate the number of engines to be started. You can also use expressions like `${n/4}` in the template to indicate the number of nodes.
3. Because `$` is a special character used by the template engine, you must escape any `$` by using `$$`. This is important when referring to environment variables in the template.
4. Any options to **ipengine** should be given in the batch script template.
5. Depending on the configuration of your system, you may have to set environment variables in the script template.

Once you have created such a script, save it with a name like `pbs.template`. Now you are ready to start your job:

```
$ ipcluster pbs -n 128 --pbs-script=pbs.template
```

Additional command line options for this mode can be found by doing:

```
$ ipcluster pbs -h
```

Using ipcluster in SSH mode

The SSH mode uses **ssh** to execute **ipengine** on remote nodes and the **ipcontroller** on localhost.

When using this mode it is highly recommended that you have set up SSH keys and are using **ssh-agent** [SSH] for password-less logins.

To use this mode you need a python file describing the cluster, here is an example of such a “clusterfile”:

```
send_furl = True
engines = { 'host1.example.com' : 2,
            'host2.example.com' : 5,
            'host3.example.com' : 1,
            'host4.example.com' : 8 }
```

Since this is a regular python file usual python syntax applies. Things to note:

- The *engines* dict, where the keys is the host we want to run engines on and the value is the number of engines to run on that host.
- `send_furl` can either be *True* or *False*, if *True* it will copy over the furl needed for **ipengine** to each host.

The `--clusterfile` command line option lets you specify the file to use for the cluster definition. Once you have your cluster file and you can **ssh** into the remote hosts without a password you are ready to start your cluster like so:

```
$ ipcluster ssh --clusterfile /path/to/my/clusterfile.py
```

Two helper shell scripts are used to start and stop **ipengine** on remote hosts:

- `sshx.sh`
- `engine_killer.sh`

Defaults for both of these are contained in the source code for **ipcluster**. The default scripts are written to a local file in a `tmap` directory and then copied to a temp directory on the remote host and executed from there. On most Unix, Linux and OS X systems this is `/tmp`.

The default `sshx.sh` is the following:

```
#!/bin/sh
"$@" &> /dev/null &
echo $!
```

If you want to use a custom `sshx.sh` script you need to use the `--sshx` option and specify the file to use. Using a custom `sshx.sh` file could be helpful when you need to setup the environment on the remote host before executing **ipengine**.

For a detailed options list:

```
$ ipcluster ssh -h
```

Current limitations of the SSH mode of **ipcluster** are:

- Untested on Windows. Would require a working **ssh** on Windows. Also, we are using shell scripts to setup and execute commands on remote hosts.
- **ipcontroller** is started on localhost, with no option to start it on a remote node.

5.2.3 Using the **ipcontroller** and **ipengine** commands

It is also possible to use the **ipcontroller** and **ipengine** commands to start your controller and engines. This approach gives you full control over all aspects of the startup process.

Starting the controller and engine on your local machine

To use **ipcontroller** and **ipengine** to start things on your local machine, do the following.

First start the controller:

```
$ ipcontroller
```

Next, start however many instances of the engine you want using (repeatedly) the command:

```
$ ipengine
```

The engines should start and automatically connect to the controller using the FURL files in `~/ipython/security`. You are now ready to use the controller and engines from IPython.

Warning: The order of the above operations is very important. You *must* start the controller before the engines, since the engines connect to the controller as they get started.

Note: On some platforms (OS X), to put the controller and engine into the background you may need to give these commands in the form `(ipcontroller &)` and `(ipengine &)` (with the parentheses) for them to work properly.

Starting the controller and engines on different hosts

When the controller and engines are running on different hosts, things are slightly more complicated, but the underlying ideas are the same:

1. Start the controller on a host using **ipcontroller**.
2. Copy `ipcontroller-engine.furl` from `~/ipython/security` on the controller's host to the host where the engines will run.
3. Use **ipengine** on the engine's hosts to start the engines.

The only thing you have to be careful of is to tell **ipengine** where the `ipcontroller-engine.furl` file is located. There are two ways you can do this:

- Put `ipcontroller-engine.furl` in the `~/ipython/security` directory on the engine's host, where it will be found automatically.
- Call **ipengine** with the `--furl-file=full_path_to_the_file` flag.

The `--furl-file` flag works like this:

```
$ ipengine --furl-file=/path/to/my/ipcontroller-engine.furl
```

Note: If the controller's and engine's hosts all have a shared file system (`~/ipython/security` is the same on all of them), then things will just work!

Make FURL files persistent

At first glance it may seem that managing the FURL files is a bit annoying. Going back to the house and key analogy, copying the FURL around each time you start the controller is like having to make a new key every time you want to unlock the door and enter your house. As with your house, you want to be able to create the key (or FURL file) once, and then simply use it at any point in the future.

This is possible, but before you do this, you **must** remove any old FURL files in the `~/ipython/security` directory.

Warning: You **must** remove old FURL files before using persistent FURL files.

Then, the only thing you have to do is decide what ports the controller will listen on for the engines and clients. This is done as follows:

```
$ ipcontroller -r --client-port=10101 --engine-port=10102
```

These options also work with all of the various modes of **ipcluster**:

```
$ ipcluster local -n 2 -r --client-port=10101 --engine-port=10102
```

Then, just copy the furl files over the first time and you are set. You can start and stop the controller and engines any many times as you want in the future, just make sure to tell the controller to use the *same* ports.

Note: You may ask the question: what ports does the controller listen on if you don't tell it to use specific ones? The default is to use high random port numbers. We do this for two reasons: i) to increase security through obscurity and ii) to multiple controllers on a given host to start and automatically use different ports.

Log files

All of the components of IPython have log files associated with them. These log files can be extremely useful in debugging problems with IPython and can be found in the directory `~/ipython/log`. Sending

the log files to us will often help us to debug any problems.

5.3 IPython's multiengine interface

The multiengine interface represents one possible way of working with a set of IPython engines. The basic idea behind the multiengine interface is that the capabilities of each engine are directly and explicitly exposed to the user. Thus, in the multiengine interface, each engine is given an id that is used to identify the engine and give it work to do. This interface is very intuitive and is designed with interactive usage in mind, and is thus the best place for new users of IPython to begin.

5.3.1 Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **ipcluster** command:

```
$ ipcluster local -n 4
```

For more detailed information about starting the controller and engines, see our [introduction](#) to using IPython for parallel computing.

5.3.2 Creating a MultiEngineClient instance

The first step is to import the IPython `IPython.kernel.client` module and then create a `MultiEngineClient` instance:

```
In [1]: from IPython.kernel import client
```

```
In [2]: mec = client.MultiEngineClient()
```

This form assumes that the `ipcontroller-mec.furl` is in the `~/ipython/security` directory on the client's host. If not, the location of the FURL file must be given as an argument to the constructor:

```
In [2]: mec = client.MultiEngineClient('/path/to/my/ipcontroller-mec.furl')
```

To make sure there are engines connected to the controller, use can get a list of engine ids:

```
In [3]: mec.get_ids()
Out[3]: [0, 1, 2, 3]
```

Here we see that there are four engines ready to do work for us.

5.3.3 Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. The multiengine interface provides two simple ways of accomplishing this: a parallel version of `map()` and `@parallel` function decorator.

Parallel map

Python's builtin `map()` functions allows a function to be applied to a sequence element-by-element. This type of code is typically trivial to parallelize. In fact, the multiengine interface in IPython already has a parallel version of `map()` that works just like its serial counterpart:

```
In [63]: serial_result = map(lambda x:x**10, range(32))

In [64]: parallel_result = mec.map(lambda x:x**10, range(32))

In [65]: serial_result==parallel_result
Out[65]: True
```

Note: The multiengine interface version of `map()` does not do any load balancing. For a load balanced version, see the task interface.

See Also:

The `map()` method has a number of options that can be controlled by the `mapper()` method. See its docstring for more information.

Parallel function decorator

Parallel functions are just like normal function, but they can be called on sequences and *in parallel*. The multiengine interface provides a decorator that turns any Python function into a parallel function:

```
In [10]: @mec.parallel()
.....: def f(x):
.....:     return 10.0*x**4
.....:

In [11]: f(range(32))      # this is done in parallel
Out[11]:
[0.0, 10.0, 160.0, ...]
```

See the docstring for the `parallel()` decorator for options.

5.3.4 Running Python commands

The most basic type of operation that can be performed on the engines is to execute Python code. Executing Python code can be done in blocking or non-blocking mode (blocking is default) using the `execute()` method.

Blocking execution

In blocking mode, the `MultiEngineClient` object (called `mec` in these examples) submits the command to the controller, which places the command in the engines' queues for execution. The `execute()` call then blocks until the engines are done executing the command:


```
# The default is to run on all engines
```

```
In [4]: mec.execute('a=5')
```

```
Out[4]:
```

```
<Results List>
```

```
[0] In [1]: a=5
```

```
[1] In [1]: a=5
```

```
[2] In [1]: a=5
```

```
[3] In [1]: a=5
```

```
In [5]: mec.execute('b=10')
```

```
Out[5]:
```

```
<Results List>
```

```
[0] In [2]: b=10
```

```
[1] In [2]: b=10
```

```
[2] In [2]: b=10
```

```
[3] In [2]: b=10
```

Python commands can be executed on specific engines by calling `execute` using the `targets` keyword argument:

```
In [6]: mec.execute('c=a+b', targets=[0, 2])
```

```
Out[6]:
```

```
<Results List>
```

```
[0] In [3]: c=a+b
```

```
[2] In [3]: c=a+b
```

```
In [7]: mec.execute('c=a-b', targets=[1, 3])
```

```
Out[7]:
```

```
<Results List>
```

```
[1] In [3]: c=a-b
```

```
[3] In [3]: c=a-b
```

```
In [8]: mec.execute('print c')
```

```
Out[8]:
```

```
<Results List>
```

```
[0] In [4]: print c
```

```
[0] Out[4]: 15
```

```
[1] In [4]: print c
```

```
[1] Out[4]: -5
```

```
[2] In [4]: print c
```

```
[2] Out[4]: 15
```

```
[3] In [4]: print c
```

```
[3] Out[4]: -5
```

This example also shows one of the most important things about the IPython engines: they have a persistent user namespaces. The `execute()` method returns a Python `dict` that contains useful information:

```
In [9]: result_dict = mec.execute('d=10; print d')

In [10]: for r in result_dict:
.....:     print r
.....:
.....:
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 0, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 1, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 2, 's
{'input': {'translated': 'd=10; print d', 'raw': 'd=10; print d'}, 'number': 5, 'id': 3, 's
```

Non-blocking execution

In non-blocking mode, `execute()` submits the command to be executed and then returns a `PendingResult` object immediately. The `PendingResult` object gives you a way of getting a result at a later time through its `get_result()` method or `r` attribute. This allows you to quickly submit long running commands without blocking your local Python/IPython session:

```
# In blocking mode
In [6]: mec.execute('import time')
Out[6]:
<Results List>
[0] In [1]: import time
[1] In [1]: import time
[2] In [1]: import time
[3] In [1]: import time

# In non-blocking mode
In [7]: pr = mec.execute('time.sleep(10)', block=False)

# Now block for the result
In [8]: pr.get_result()
Out[8]:
<Results List>
[0] In [2]: time.sleep(10)
[1] In [2]: time.sleep(10)
[2] In [2]: time.sleep(10)
[3] In [2]: time.sleep(10)

# Again in non-blocking mode
In [9]: pr = mec.execute('time.sleep(10)', block=False)

# Poll to see if the result is ready
In [10]: pr.get_result(block=False)

# A shorthand for get_result(block=True)
In [11]: pr.r
Out[11]:
<Results List>
[0] In [3]: time.sleep(10)
[1] In [3]: time.sleep(10)
```

```
[2] In [3]: time.sleep(10)
[3] In [3]: time.sleep(10)
```

Often, it is desirable to wait until a set of `PendingResult` objects are done. For this, there is a the method `barrier()`. This method takes a tuple of `PendingResult` objects and blocks until all of the associated results are ready:

```
In [72]: mec.block=False

# A trivial list of PendingResults objects
In [73]: pr_list = [mec.execute('time.sleep(3)') for i in range(10)]

# Wait until all of them are done
In [74]: mec.barrier(pr_list)

# Then, their results are ready using get_result or the r attribute
In [75]: pr_list[0].r
Out[75]:
<Results List>
[0] In [20]: time.sleep(3)
[1] In [19]: time.sleep(3)
[2] In [20]: time.sleep(3)
[3] In [19]: time.sleep(3)
```

The `block` and `targets` keyword arguments and attributes

Most methods in the multiengine interface (like `execute()`) accept `block` and `targets` as keyword arguments. As we have seen above, these keyword arguments control the blocking mode and which engines the command is applied to. The `MultiEngineClient` class also has `block` and `targets` attributes that control the default behavior when the keyword arguments are not provided. Thus the following logic is used for `block` and `targets`:

- If no keyword argument is provided, the instance attributes are used.
- Keyword argument, if provided override the instance attributes.

The following examples demonstrate how to use the instance attributes:

```
In [16]: mec.targets = [0,2]

In [17]: mec.block = False

In [18]: pr = mec.execute('a=5')

In [19]: pr.r
Out[19]:
<Results List>
[0] In [6]: a=5
[2] In [6]: a=5

# Note targets='all' means all engines
In [20]: mec.targets = 'all'
```

```
In [21]: mec.block = True
```

```
In [22]: mec.execute('b=10; print b')
```

```
Out[22]:
```

```
<Results List>
```

```
[0] In [7]: b=10; print b
```

```
[0] Out[7]: 10
```

```
[1] In [6]: b=10; print b
```

```
[1] Out[6]: 10
```

```
[2] In [7]: b=10; print b
```

```
[2] Out[7]: 10
```

```
[3] In [6]: b=10; print b
```

```
[3] Out[6]: 10
```

The `block` and `targets` instance attributes also determine the behavior of the parallel magic commands.

Parallel magic commands

We provide a few IPython magic commands (`%px`, `%autopx` and `%result`) that make it more pleasant to execute Python commands on the engines interactively. These are simply shortcuts to `execute()` and `get_result()`. The `%px` magic executes a single Python command on the engines specified by the `targets` attribute of the `MultiEngineClient` instance (by default this is `'all'`):

```
# Make this MultiEngineClient active for parallel magic commands
```

```
In [23]: mec.activate()
```

```
In [24]: mec.block=True
```

```
In [25]: import numpy
```

```
In [26]: %px import numpy
```

```
Executing command on Controller
```

```
Out[26]:
```

```
<Results List>
```

```
[0] In [8]: import numpy
```

```
[1] In [7]: import numpy
```

```
[2] In [8]: import numpy
```

```
[3] In [7]: import numpy
```

```
In [27]: %px a = numpy.random.rand(2,2)
```

```
Executing command on Controller
```

```
Out[27]:
```

```
<Results List>
```

```
[0] In [9]: a = numpy.random.rand(2,2)
```

```
[1] In [8]: a = numpy.random.rand(2,2)
```

```
[2] In [9]: a = numpy.random.rand(2,2)
```

```
[3] In [8]: a = numpy.random.rand(2,2)
```

```

In [28]: %px print numpy.linalg.eigvals(a)
Executing command on Controller
Out[28]:
<Results List>
[0] In [10]: print numpy.linalg.eigvals(a)
[0] Out[10]: [ 1.28167017  0.14197338]

[1] In [9]: print numpy.linalg.eigvals(a)
[1] Out[9]: [-0.14093616  1.27877273]

[2] In [10]: print numpy.linalg.eigvals(a)
[2] Out[10]: [-0.37023573  1.06779409]

[3] In [9]: print numpy.linalg.eigvals(a)
[3] Out[9]: [ 0.83664764 -0.25602658]

```

The `%result` magic gets and prints the `stdin/stdout/stderr` of the last command executed on each engine. It is simply a shortcut to the `get_result()` method:

```

In [29]: %result
Out[29]:
<Results List>
[0] In [10]: print numpy.linalg.eigvals(a)
[0] Out[10]: [ 1.28167017  0.14197338]

[1] In [9]: print numpy.linalg.eigvals(a)
[1] Out[9]: [-0.14093616  1.27877273]

[2] In [10]: print numpy.linalg.eigvals(a)
[2] Out[10]: [-0.37023573  1.06779409]

[3] In [9]: print numpy.linalg.eigvals(a)
[3] Out[9]: [ 0.83664764 -0.25602658]

```

The `%autopx` magic switches to a mode where everything you type is executed on the engines given by the `targets` attribute:

```
In [30]: mec.block=False
```

```

In [31]: %autopx
Auto Parallel Enabled
Type %autopx to disable

```

```

In [32]: max_evals = []
<IPython.kernel.multiengineclient.PendingResult object at 0x17b8a70>

```

```

In [33]: for i in range(100):
.....:     a = numpy.random.rand(10,10)
.....:     a = a+a.transpose()
.....:     evals = numpy.linalg.eigvals(a)
.....:     max_evals.append(evals[0].real)
.....:

```

```
.....  
<IPython.kernel.multiengineclient.PendingResult object at 0x17af8f0>  
  
In [34]: %autopx  
Auto Parallel Disabled  
  
In [35]: mec.block=True  
  
In [36]: px print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)  
Executing command on Controller  
Out[36]:  
<Results List>  
[0] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)  
[0] Out[13]: Average max eigenvalue is: 10.1387247332  
  
[1] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)  
[1] Out[12]: Average max eigenvalue is: 10.2076902286  
  
[2] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)  
[2] Out[13]: Average max eigenvalue is: 10.1891484655  
  
[3] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)  
[3] Out[12]: Average max eigenvalue is: 10.1158837784
```

5.3.5 Moving Python objects around

In addition to executing code on engines, you can transfer Python objects to and from your IPython session and the engines. In IPython, these operations are called `push()` (sending an object to the engines) and `pull()` (getting an object from the engines).

Basic push and pull

Here are some examples of how you use `push()` and `pull()`:

```
In [38]: mec.push(dict(a=1.03234,b=3453))  
Out[38]: [None, None, None, None]  
  
In [39]: mec.pull('a')  
Out[39]: [1.03234, 1.03234, 1.03234, 1.03234]  
  
In [40]: mec.pull('b',targets=0)  
Out[40]: [3453]  
  
In [41]: mec.pull(('a','b'))  
Out[41]: [[1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453]]  
  
In [42]: mec.zip_pull(('a','b'))  
Out[42]: [(1.03234, 1.03234, 1.03234, 1.03234), (3453, 3453, 3453, 3453)]  
  
In [43]: mec.push(dict(c='speed'))  
Out[43]: [None, None, None, None]
```

```
In [44]: %px print c
Executing command on Controller
Out[44]:
<Results List>
[0] In [14]: print c
[0] Out[14]: speed

[1] In [13]: print c
[1] Out[13]: speed

[2] In [14]: print c
[2] Out[14]: speed

[3] In [13]: print c
[3] Out[13]: speed
```

In non-blocking mode `push()` and `pull()` also return `PendingResult` objects:

```
In [47]: mec.block=False

In [48]: pr = mec.pull('a')

In [49]: pr.r
Out[49]: [1.03234, 1.03234, 1.03234, 1.03234]
```

Push and pull for functions

Functions can also be pushed and pulled using `push_function()` and `pull_function()`:

```
In [52]: mec.block=True

In [53]: def f(x):
.....:     return 2.0*x**4
.....:

In [54]: mec.push_function(dict(f=f))
Out[54]: [None, None, None, None]

In [55]: mec.execute('y = f(4.0)')
Out[55]:
<Results List>
[0] In [15]: y = f(4.0)
[1] In [14]: y = f(4.0)
[2] In [15]: y = f(4.0)
[3] In [14]: y = f(4.0)

In [56]: %px print y
Executing command on Controller
Out[56]:
<Results List>
[0] In [16]: print y
```

```
[0] Out[16]: 512.0

[1] In [15]: print y
[1] Out[15]: 512.0

[2] In [16]: print y
[2] Out[16]: 512.0

[3] In [15]: print y
[3] Out[15]: 512.0
```

Dictionary interface

As a shorthand to `push()` and `pull()`, the `MultiEngineClient` class implements some of the Python dictionary interface. This make the remote namespaces of the engines appear as a local dictionary. Underneath, this uses `push()` and `pull()`:

```
In [50]: mec.block=True

In [51]: mec['a']=['foo','bar']

In [52]: mec['a']
Out[52]: [['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar'], ['foo', 'bar']]
```

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `MultiEngineClient` class, `scatter()` is from the interactive IPython session to the engines and `gather()` is from the engines back to the interactive IPython session. For scatter/gather operations between engines, MPI should be used:

```
In [58]: mec.scatter('a', range(16))
Out[58]: [None, None, None, None]
```

```
In [59]: px print a
Executing command on Controller
Out[59]:
<Results List>
[0] In [17]: print a
[0] Out[17]: [0, 1, 2, 3]

[1] In [16]: print a
[1] Out[16]: [4, 5, 6, 7]

[2] In [17]: print a
[2] Out[17]: [8, 9, 10, 11]

[3] In [16]: print a
[3] Out[16]: [12, 13, 14, 15]
```



```
In [60]: mec.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

5.3.6 Other things to look at

How to do parallel list comprehensions

In many cases list comprehensions are nicer than using the map function. While we don't have fully parallel list comprehensions, it is simple to get the basic effect using `scatter()` and `gather()`:

```
In [66]: mec.scatter('x', range(64))
Out[66]: [None, None, None, None]
```

```
In [67]: px y = [i**10 for i in x]
Executing command on Controller
Out[67]:
<Results List>
[0] In [19]: y = [i**10 for i in x]
[1] In [18]: y = [i**10 for i in x]
[2] In [19]: y = [i**10 for i in x]
[3] In [18]: y = [i**10 for i in x]
```

```
In [68]: y = mec.gather('y')
```

```
In [69]: print y
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, ...]
```

Parallel exceptions

In the multiengine interface, parallel commands can raise Python exceptions, just like serial commands. But, it is a little subtle, because a single parallel command can actually raise multiple exceptions (one for each engine the command was run on). To express this idea, the MultiEngine interface has a `CompositeError` exception class that will be raised in most cases. The `CompositeError` class is a special type of exception that wraps one or more other types of exceptions. Here is how it works:

```
In [76]: mec.block=True
```

```
In [77]: mec.execute('1/0')
```

```
-----
CompositeError                                Traceback (most recent call last)
```

```
/ipython1-client-r3021/docs/examples/<ipython console> in <module>()
```

```
/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in execute(self, lines, target
```

```
    432         targets, block = self._findTargetsAndBlock(targets, block)
```

```
    433         result = blockingCallFromThread(self.smultiengine.execute, lines,
```

```
--> 434         targets=targets, block=block)
```

```
435         if block:
436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.py in blockingCallFromThread(f, *a, **k)
72         result.raiseException()
73     except Exception, e:
--> 74         raise e
75     return result
76
```

```
CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero
```

Notice how the error message printed when `CompositeError` is raised has information about the individual exceptions that were raised on each engine. If you want, you can even raise one of these original exceptions:

```
In [80]: try:
.....:     mec.execute('1/0')
.....: except client.CompositeError, e:
.....:     e.raise_exception()
.....:
.....:

-----
ZeroDivisionError                                Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/error.py in raise_exception(self, excid)
156         raise IndexError("an exception with index %i does not exist"%excid)
157     else:
--> 158         raise et, ev, etb
159
160 def collect_exceptions(rlist, method):
```

```
ZeroDivisionError: integer division or modulo by zero
```

If you are working in IPython, you can simple type `%debug` after one of these `CompositeError` exceptions is raised, and inspect the exception instance:

```
In [81]: mec.execute('1/0')

-----
CompositeError                                Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.py in execute(self, lines, target
432         targets, block = self._findTargetsAndBlock(targets, block)
433         result = blockingCallFromThread(self.smultiengine.execute, lines,
--> 434         targets=targets, block=block)
```

```

435         if block:
436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k)
    72         result.raiseException()
    73     except Exception, e:
--> 74         raise e
    75     return result
    76

```

```

CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero

```

In [82]: %debug

>

```

/ipython1-client-r3021/ipython1/kernel/twistedutil.py(74)blockingCallFromThread()
    73         except Exception, e:
--> 74             raise e
    75     return result

```

```

ipdb> e.
e.__class__          e.__getitem__        e.__new__            e.__setstate__      e.args
e.__delattr__        e.__getslice__       e.__reduce__         e.__str__           e.elist
e.__dict__           e.__hash__           e.__reduce_ex__      e.__weakref__       e.message
e.__doc__            e.__init__           e.__repr__           e.__get_engine_str  e.print_traceback
e.__getattribute__   e.__module__         e.__setattr__        e.__get_traceback   e.raise_exception
ipdb> e.print_tracebacks()
[0:execute]:

```

```

-----
ZeroDivisionError                                Traceback (most recent call last)

```

```

/ipython1-client-r3021/docs/examples/<string> in <module>()

```

```

ZeroDivisionError: integer division or modulo by zero

```

```

[1:execute]:

```

```

-----
ZeroDivisionError                                Traceback (most recent call last)

```

```

/ipython1-client-r3021/docs/examples/<string> in <module>()

```

```

ZeroDivisionError: integer division or modulo by zero

```

```

[2:execute]:

```

```

-----
ZeroDivisionError                                Traceback (most recent call last)

```

```

/ipython1-client-r3021/docs/examples/<string> in <module>()

```

ZeroDivisionError: integer division or modulo by zero

```
[3:execute]:
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
```

```
/ipython1-client-r3021/docs/examples/<string> in <module>()
```

ZeroDivisionError: integer division or modulo by zero

Note: The above example appears to be broken right now because of a change in how we are using Twisted.

All of this same error handling magic even works in non-blocking mode:

```
In [83]: mec.block=False
```

```
In [84]: pr = mec.execute('1/0')
```

```
In [85]: pr.r
```

```
-----
CompositeError                                Traceback (most recent call last)
```

```
/ipython1-client-r3021/docs/examples/<ipython console> in <module>()
```

```
/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in _get_r(self)
```

```
170
171     def _get_r(self):
--> 172         return self.get_result(block=True)
173
174     r = property(_get_r)
```

```
/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_result(self, default, block)
```

```
131         return self.result
132         try:
--> 133             result = self.client.get_pending_deferred(self.result_id, block)
134             except error.ResultNotCompleted:
135                 return default
```

```
/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_pending_deferred(self, result_id, block)
```

```
385
386     def get_pending_deferred(self, deferredID, block):
--> 387         return blockingCallFromThread(self.smultiengine.get_pending_deferred, deferredID, block)
388
389     def barrier(self, pendingResults):
```

```
/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k)
```

```
72         result.raiseException()
73         except Exception, e:
--> 74             raise e
75         return result
76
```

```
CompositeError: one or more exceptions from call to method: execute
[0:execute]: ZeroDivisionError: integer division or modulo by zero
[1:execute]: ZeroDivisionError: integer division or modulo by zero
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero
```

5.4 The IPython task interface

The task interface to the controller presents the engines as a fault tolerant, dynamic load-balanced system or workers. Unlike the multiengine interface, in the task interface, the user have no direct access to individual engines. In some ways, this interface is simpler, but in other ways it is more powerful.

Best of all the user can use both of these interfaces running at the same time to take advantage or both of their strengths. When the user can break up the user's work into segments that do not depend on previous execution, the task interface is ideal. But it also has more power and flexibility, allowing the user to guide the distribution of jobs, without having to assign tasks to engines explicitly.

5.4.1 Starting the IPython controller and engines

To follow along with this tutorial, you will need to start the IPython controller and four IPython engines. The simplest way of doing this is to use the **ipcluster** command:

```
$ ipcluster local -n 4
```

For more detailed information about starting the controller and engines, see our [introduction](#) to using IPython for parallel computing.

5.4.2 Creating a TaskClient instance

The first step is to import the IPython `IPython.kernel.client` module and then create a `TaskClient` instance:

```
In [1]: from IPython.kernel import client
```

```
In [2]: tc = client.TaskClient()
```

This form assumes that the `ipcontroller-tc.furl` is in the `~/ipython/security` directory on the client's host. If not, the location of the FURL file must be given as an argument to the constructor:

```
In [2]: mec = client.TaskClient('/path/to/my/ipcontroller-tc.furl')
```

5.4.3 Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. Like the multiengine interface, the task interface provides two simple ways of accomplishing this: a parallel version of `map()` and `@parallel` function decorator. However, the verions in the task interface have

one important difference: they are dynamically load balanced. Thus, if the execution time per item varies significantly, you should use the versions in the task interface.

Parallel map

The `parallel map()` in the task interface is similar to that in the multiengine interface:

```
In [63]: serial_result = map(lambda x:x**10, range(32))
In [64]: parallel_result = tc.map(lambda x:x**10, range(32))
In [65]: serial_result==parallel_result
Out[65]: True
```

Parallel function decorator

Parallel functions are just like normal function, but they can be called on sequences and *in parallel*. The multiengine interface provides a decorator that turns any Python function into a parallel function:

```
In [10]: @tc.parallel()
.....: def f(x):
.....:     return 10.0*x**4
.....:

In [11]: f(range(32))      # this is done in parallel
Out[11]:
[0.0, 10.0, 160.0, ...]
```

5.4.4 More details

The `TaskClient` has many more powerful features that allow quite a bit of flexibility in how tasks are defined and run. The next places to look are in the following classes:

- `IPython.kernel.client.TaskClient`
- `IPython.kernel.client.StringTask`
- `IPython.kernel.client.MapTask`

The following is an overview of how to use these classes together:

1. Create a `TaskClient`.
2. Create one or more instances of `StringTask` or `MapTask` to define your tasks.
3. Submit your tasks to using the `run()` method of your `TaskClient` instance.
4. Use `TaskClient.get_task_result()` to get the results of the tasks.

We are in the process of developing more detailed information about the task interface. For now, the docstrings of the `TaskClient`, `StringTask` and `MapTask` classes should be consulted.

5.5 Using MPI with IPython

Often, a parallel algorithm will require moving data between the engines. One way of accomplishing this is by doing a pull and then a push using the multiengine client. However, this will be slow as all the data has to go through the controller to the client and then back through the controller, to its final destination.

A much better way of moving data between engines is to use a message passing library, such as the Message Passing Interface (MPI) [MPI]. IPython's parallel computing architecture has been designed from the ground up to integrate with MPI. This document describes how to use MPI with IPython.

5.5.1 Additional installation requirements

If you want to use MPI with IPython, you will need to install:

- A standard MPI implementation such as OpenMPI [OpenMPI] or MPICH.
- The mpi4py [mpi4py] package.

Note: The mpi4py package is not a strict requirement. However, you need to have *some* way of calling MPI from Python. You also need some way of making sure that `MPI_Init()` is called when the IPython engines start up. There are a number of ways of doing this and a good number of associated subtleties. We highly recommend just using mpi4py as it takes care of most of these problems. If you want to do something different, let us know and we can help you get started.

5.5.2 Starting the engines with MPI enabled

To use code that calls MPI, there are typically two things that MPI requires.

1. The process that wants to call MPI must be started using **mpiexec** or a batch system (like PBS) that has MPI support.
2. Once the process starts, it must call `MPI_Init()`.

There are a couple of ways that you can start the IPython engines and get these things to happen.

Automatic starting using mpiexec and ipcluster

The easiest approach is to use the *mpiexec* mode of **ipcluster**, which will first start a controller and then a set of engines using **mpiexec**:

```
$ ipcluster mpiexec -n 4
```

This approach is best as interrupting **ipcluster** will automatically stop and clean up the controller and engines.

Manual starting using mpiexec

If you want to start the IPython engines using the **mpiexec**, just do:

```
$ mpiexec -n 4 ipengine --mpi=mpi4py
```

This requires that you already have a controller running and that the FURL files for the engines are in place. We also have built in support for PyTrilinos [PyTrilinos], which can be used (assuming is installed) by starting the engines with:

```
mpiexec -n 4 ipengine --mpi=pytrilinos
```

Automatic starting using PBS and ipcluster

The **ipcluster** command also has built-in integration with PBS. For more information on this approach, see our documentation on *ipcluster*.

5.5.3 Actually using MPI

Once the engines are running with MPI enabled, you are ready to go. You can now call any code that uses MPI in the IPython engines. And, all of this can be done interactively. Here we show a simple example that uses mpi4py [mpi4py] version 1.1.0 or later.

First, let's define a simple function that uses MPI to calculate the sum of a distributed array. Save the following text in a file called `psum.py`:

```
from mpi4py import MPI
import numpy as np

def psum(a):
    s = np.sum(a)
    rcvBuf = np.array(0.0, 'd')
    MPI.COMM_WORLD.Allreduce([s, MPI.DOUBLE],
                             [rcvBuf, MPI.DOUBLE],
                             op=MPI.SUM)
    return rcvBuf
```

Now, start an IPython cluster in the same directory as `psum.py`:

```
$ ipcluster mpiexec -n 4
```

Finally, connect to the cluster and use this function interactively. In this case, we create a random array on each engine and sum up all the random arrays using our `psum()` function:

```
In [1]: from IPython.kernel import client

In [2]: mec = client.MultiEngineClient()

In [3]: mec.activate()

In [4]: px import numpy as np
```


Parallel execution on engines: all

Out[4]:

<Results List>

```
[0] In [13]: import numpy as np
[1] In [13]: import numpy as np
[2] In [13]: import numpy as np
[3] In [13]: import numpy as np
```

In [6]: px a = np.random.rand(100)

Parallel execution on engines: all

Out[6]:

<Results List>

```
[0] In [15]: a = np.random.rand(100)
[1] In [15]: a = np.random.rand(100)
[2] In [15]: a = np.random.rand(100)
[3] In [15]: a = np.random.rand(100)
```

In [7]: px from psum import psum

Parallel execution on engines: all

Out[7]:

<Results List>

```
[0] In [16]: from psum import psum
[1] In [16]: from psum import psum
[2] In [16]: from psum import psum
[3] In [16]: from psum import psum
```

In [8]: px s = psum(a)

Parallel execution on engines: all

Out[8]:

<Results List>

```
[0] In [17]: s = psum(a)
[1] In [17]: s = psum(a)
[2] In [17]: s = psum(a)
[3] In [17]: s = psum(a)
```

In [9]: px print s

Parallel execution on engines: all

Out[9]:

<Results List>

```
[0] In [18]: print s
[0] Out[18]: 187.451545803
```

```
[1] In [18]: print s
[1] Out[18]: 187.451545803
```

```
[2] In [18]: print s
[2] Out[18]: 187.451545803
```

```
[3] In [18]: print s
[3] Out[18]: 187.451545803
```

Any Python code that makes calls to MPI can be used in this manner, including compiled C, C++ and Fortran libraries that have been exposed to Python.

5.6 Security details of IPython

IPython's `IPython.kernel` package exposes the full power of the Python interpreter over a TCP/IP network for the purposes of parallel computing. This feature brings up the important question of IPython's security model. This document gives details about this model and how it is implemented in IPython's architecture.

5.6.1 Processs and network topology

To enable parallel computing, IPython has a number of different processes that run. These processes are discussed at length in the IPython documentation and are summarized here:

- The IPython *engine*. This process is a full blown Python interpreter in which user code is executed. Multiple engines are started to make parallel computing possible.
- The IPython *controller*. This process manages a set of engines, maintaining a queue for each and presenting an asynchronous interface to the set of engines.
- The IPython *client*. This process is typically an interactive Python process that is used to coordinate the engines to get a parallel computation done.

Collectively, these three processes are called the IPython *kernel*.

These three processes communicate over TCP/IP connections with a well defined topology. The IPython controller is the only process that listens on TCP/IP sockets. Upon starting, an engine connects to a controller and registers itself with the controller. These engine/controller TCP/IP connections persist for the lifetime of each engine.

The IPython client also connects to the controller using one or more TCP/IP connections. These connections persist for the lifetime of the client only.

A given IPython controller and set of engines typically has a relatively short lifetime. Typically this lifetime corresponds to the duration of a single parallel simulation performed by a single user. Finally, the controller, engines and client processes typically execute with the permissions of that same user. More specifically, the controller and engines are *not* executed as root or with any other superuser permissions.

5.6.2 Application logic

When running the IPython kernel to perform a parallel computation, a user utilizes the IPython client to send Python commands and data through the IPython controller to the IPython engines, where those commands are executed and the data processed. The design of IPython ensures that the client is the only access point for the capabilities of the engines. That is, the only way of addressing the engines is through a client.

A user can utilize the client to instruct the IPython engines to execute arbitrary Python commands. These Python commands can include calls to the system shell, access the filesystem, etc., as required by the user's application code. From this perspective, when a user runs an IPython engine on a host, that engine has the same capabilities and permissions as the user themselves (as if they were logged onto the engine's host with a terminal).

5.6.3 Secure network connections

Overview

All TCP/IP connections between the client and controller as well as the engines and controller are fully encrypted and authenticated. This section describes the details of the encryption and authentication approached used within IPython.

IPython uses the Foolscape network protocol [\[Foolscape\]](#) for all communications between processes. Thus, the details of IPython's security model are directly related to those of Foolscape. Thus, much of the following discussion is actually just a discussion of the security that is built in to Foolscape.

Encryption

For encryption purposes, IPython and Foolscape use the well known Secure Socket Layer (SSL) protocol [\[RFC5246\]](#). We use the implementation of this protocol provided by the OpenSSL project through the pyOpenSSL [\[pyOpenSSL\]](#) Python bindings to OpenSSL.

Authentication

IPython clients and engines must also authenticate themselves with the controller. This is handled in a capabilities based security model [\[Capability\]](#). In this model, the controller creates a strong cryptographic key or token that represents each set of capability that the controller offers. Any party who has this key and presents it to the controller has full access to the corresponding capabilities of the controller. This model is analogous to using a physical key to gain access to physical items (capabilities) behind a locked door.

For a capabilities based authentication system to prevent unauthorized access, two things must be ensured:

- The keys must be cryptographically strong. Otherwise attackers could gain access by a simple brute force key guessing attack.
- The actual keys must be distributed only to authorized parties.

The keys in Foolscape are called Foolscape URL's or FURLs. The following section gives details about how these FURLs are created in Foolscape. The IPython controller creates a number of FURLs for different purposes:

- One FURL that grants IPython engines access to the controller. Also implicit in this access is permission to execute code sent by an authenticated IPython client.
- Two or more FURLs that grant IPython clients access to the controller. Implicit in this access is permission to give the controller's engine code to execute.

Upon starting, the controller creates these different FURLS and writes them files in the user-read-only directory `$HOME/.ipython/security`. Thus, only the user who starts the controller has access to the FURLs.

For an IPython client or engine to authenticate with a controller, it must present the appropriate FURL to the controller upon connecting. If the FURL matches what the controller expects for a given capability, access is granted. If not, access is denied. The exchange of FURLs is done after encrypted communications channels have been established to prevent attackers from capturing them.

Note: The FURL is similar to an unsigned private key in SSH.

Details of the Foolscape handshake

In this section we detail the precise security handshake that takes place at the beginning of any network connection in IPython. For the purposes of this discussion, the SERVER is the IPython controller process and the CLIENT is the IPython engine or client process.

Upon starting, all IPython processes do the following:

1. Create a public key x509 certificate (ISO/IEC 9594).
2. Create a hash of the contents of the certificate using the SHA-1 algorithm. The base-32 encoded version of this hash is saved by the process as its process id (actually in Foolscape, this is the Tub id, but here refer to it as the process id).

Upon starting, the IPython controller also does the following:

1. Save the x509 certificate to disk in a secure location. The CLIENT certificate is never saved to disk.
2. Create a FURL for each capability that the controller has. There are separate capabilities the controller offers for clients and engines. The FURL is created using: a) the process id of the SERVER, b) the IP address and port the SERVER is listening on and c) a 160 bit, cryptographically secure string that represents the capability (the “capability id”).
3. The FURLs are saved to disk in a secure location on the SERVER’s host.

For a CLIENT to be able to connect to the SERVER and access a capability of that SERVER, the CLIENT must have knowledge of the FURL for that SERVER’s capability. This typically requires that the file containing the FURL be moved from the SERVER’s host to the CLIENT’s host. This is done by the end user who started the SERVER and wishes to have a CLIENT connect to the SERVER.

When a CLIENT connects to the SERVER, the following handshake protocol takes place:

1. The CLIENT tells the SERVER what process (or Tub) id it expects the SERVER to have.
2. If the SERVER has that process id, it notifies the CLIENT that it will now enter encrypted mode. If the SERVER has a different id, the SERVER aborts.
3. Both CLIENT and SERVER initiate the SSL handshake protocol.
4. Both CLIENT and SERVER request the certificate of their peer and verify that certificate. If this succeeds, all further communications are encrypted.
5. Both CLIENT and SERVER send a hello block containing connection parameters and their process id.
6. The CLIENT and SERVER check that their peer’s stated process id matches the hash of the x509 certificate the peer presented. If not, the connection is aborted.
7. The CLIENT verifies that the SERVER’s stated id matches the id of the SERVER the CLIENT is intending to connect to. If not, the connection is aborted.

8. The CLIENT and SERVER elect a master who decides on the final connection parameters.

The public/private key pair associated with each process's x509 certificate are completely hidden from this handshake protocol. There are however, used internally by OpenSSL as part of the SSL handshake protocol. Each process keeps their own private key hidden and sends its peer only the public key (embedded in the certificate).

Finally, when the CLIENT requests access to a particular SERVER capability, the following happens:

1. The CLIENT asks the SERVER for access to a capability by presenting that capabilities id.
2. If the SERVER has a capability with that id, access is granted. If not, access is not granted.
3. Once access has been gained, the CLIENT can use the capability.

5.6.4 Specific security vulnerabilities

There are a number of potential security vulnerabilities present in IPython's architecture. In this section we discuss those vulnerabilities and detail how the security architecture described above prevents them from being exploited.

Unauthorized clients

The IPython client can instruct the IPython engines to execute arbitrary Python code with the permissions of the user who started the engines. If an attacker were able to connect their own hostile IPython client to the IPython controller, they could instruct the engines to execute code.

This attack is prevented by the capabilities based client authentication performed after the encrypted channel has been established. The relevant authentication information is encoded into the FURL that clients must present to gain access to the IPython controller. By limiting the distribution of those FURLs, a user can grant access to only authorized persons.

It is highly unlikely that a client FURL could be guessed by an attacker in a brute force guessing attack. A given instance of the IPython controller only runs for a relatively short amount of time (on the order of hours). Thus an attacker would have only a limited amount of time to test a search space of size 2^{32} . Furthermore, even if a controller were to run for a longer amount of time, this search space is quite large (larger for instance than that of typical username/password pair).

Unauthorized engines

If an attacker were able to connect a hostile engine to a user's controller, the user might unknowingly send sensitive code or data to the hostile engine. This attacker's engine would then have full access to that code and data.

This type of attack is prevented in the same way as the unauthorized client attack, through the usage of the capabilities based authentication scheme.

Unauthorized controllers

It is also possible that an attacker could try to convince a user's IPython client or engine to connect to a hostile IPython controller. That controller would then have full access to the code and data sent between the IPython client and the IPython engines.

Again, this attack is prevented through the FURLs, which ensure that a client or engine connects to the correct controller. It is also important to note that the FURLs also encode the IP address and port that the controller is listening on, so there is little chance of mistakenly connecting to a controller running on a different IP address and port.

When starting an engine or client, a user must specify which FURL to use for that connection. Thus, in order to introduce a hostile controller, the attacker must convince the user to use the FURLs associated with the hostile controller. As long as a user is diligent in only using FURLs from trusted sources, this attack is not possible.

5.6.5 Other security measures

A number of other measures are taken to further limit the security risks involved in running the IPython kernel.

First, by default, the IPython controller listens on random port numbers. While this can be overridden by the user, in the default configuration, an attacker would have to do a port scan to even find a controller to attack. When coupled with the relatively short running time of a typical controller (on the order of hours), an attacker would have to work extremely hard and extremely *fast* to even find a running controller to attack.

Second, much of the time, especially when run on supercomputers or clusters, the controller is running behind a firewall. Thus, for engines or client to connect to the controller:

- The different processes have to all be behind the firewall.

or:

- The user has to use SSH port forwarding to tunnel the connections through the firewall.

In either case, an attacker is presented with addition barriers that prevent attacking or even probing the system.

5.6.6 Summary

IPython's architecture has been carefully designed with security in mind. The capabilities based authentication model, in conjunction with the encrypted TCP/IP channels, address the core potential vulnerabilities in the system, while still enabling user's to use the system in open networks.

5.6.7 Other questions

About keys

Can you clarify the roles of the certificate and its keys versus the FURL, which is also called a key?

The certificate created by IPython processes is a standard public key x509 certificate, that is used by the SSL handshake protocol to setup encrypted channel between the controller and the IPython engine or client. This public and private key associated with this certificate are used only by the SSL handshake protocol in setting up this encrypted channel.

The FURL serves a completely different and independent purpose from the key pair associated with the certificate. When we refer to a FURL as a key, we are using the word “key” in the capabilities based security model sense. This has nothing to do with “key” in the public/private key sense used in the SSL protocol.

With that said the FURL is used as an cryptographic key, to grant IPython engines and clients access to particular capabilities that the controller offers.

Self signed certificates

Is the controller creating a self-signed certificate? Is this created for per instance/session, one-time-setup or each-time the controller is started?

The Foolscape network protocol, which handles the SSL protocol details, creates a self-signed x509 certificate using OpenSSL for each IPython process. The lifetime of the certificate is handled differently for the IPython controller and the engines/client.

For the IPython engines and client, the certificate is only held in memory for the lifetime of its process. It is never written to disk.

For the controller, the certificate can be created anew each time the controller starts or it can be created once and reused each time the controller starts. If at any point, the certificate is deleted, a new one is created the next time the controller starts.

SSL private key

How the private key (associated with the certificate) is distributed?

In the usual implementation of the SSL protocol, the private key is never distributed. We follow this standard always.

SSL versus Foolscape authentication

Many SSL connections only perform one sided authentication (the server to the client). How is the client authentication in IPython’s system related to SSL authentication?

We perform a two way SSL handshake in which both parties request and verify the certificate of their peer. This mutual authentication is handled by the SSL handshake and is separate and independent from the additional authentication steps that the CLIENT and SERVER perform after an encrypted channel is established.

5.7 Getting started with Windows HPC Server 2008

5.7.1 Introduction

The Python programming language is an increasingly popular language for numerical computing. This is due to a unique combination of factors. First, Python is a high-level and *interactive* language that is well matched to interactive numerical work. Second, it is easy (often times trivial) to integrate legacy C/C++/Fortran code into Python. Third, a large number of high-quality open source projects provide all the needed building blocks for numerical computing: numerical arrays (NumPy), algorithms (SciPy), 2D/3D Visualization (Matplotlib, Mayavi, Chaco), Symbolic Mathematics (Sage, Sympy) and others.

The IPython project is a core part of this open-source toolchain and is focused on creating a comprehensive environment for interactive and exploratory computing in the Python programming language. It enables all of the above tools to be used interactively and consists of two main components:

- An enhanced interactive Python shell with support for interactive plotting and visualization.
- An architecture for interactive parallel computing.

With these components, it is possible to perform all aspects of a parallel computation interactively. This type of workflow is particularly relevant in scientific and numerical computing where algorithms, code and data are continually evolving as the user/developer explores a problem. The broad trends in computing (commodity clusters, multicore, cloud computing, etc.) make these capabilities of IPython particularly relevant.

While IPython is a cross platform tool, it has particularly strong support for Windows based compute clusters running Windows HPC Server 2008. This document describes how to get started with IPython on Windows HPC Server 2008. The content and emphasis here is practical: installing IPython, configuring IPython to use the Windows job scheduler and running example parallel programs interactively. A more complete description of IPython's parallel computing capabilities can be found in IPython's online documentation (<http://ipython.scipy.org/moin/Documentation>).

5.7.2 Setting up your Windows cluster

This document assumes that you already have a cluster running Windows HPC Server 2008. Here is a broad overview of what is involved with setting up such a cluster:

1. Install Windows Server 2008 on the head and compute nodes in the cluster.
2. Setup the network configuration on each host. Each host should have a static IP address.
3. On the head node, activate the "Active Directory Domain Services" role and make the head node the domain controller.
4. Join the compute nodes to the newly created Active Directory (AD) domain.
5. Setup user accounts in the domain with shared home directories.
6. Install the HPC Pack 2008 on the head node to create a cluster.
7. Install the HPC Pack 2008 on the compute nodes.

More details about installing and configuring Windows HPC Server 2008 can be found on the Windows HPC Home Page (<http://www.microsoft.com/hpc>). Regardless of what steps you follow to set up your cluster, the remainder of this document will assume that:

- There are domain users that can log on to the AD domain and submit jobs to the cluster scheduler.
- These domain users have shared home directories. While shared home directories are not required to use IPython, they make it much easier to use IPython.

5.7.3 Installation of IPython and its dependencies

IPython and all of its dependencies are freely available and open source. These packages provide a powerful and cost-effective approach to numerical and scientific computing on Windows. The following dependencies are needed to run IPython on Windows:

- Python 2.5 or 2.6 (<http://www.python.org>)
- pywin32 (<http://sourceforge.net/projects/pywin32/>)
- PyReadline (<https://launchpad.net/pyreadline>)
- zope.interface and Twisted (<http://twistedmatrix.com>)
- Foolcap (<http://foolscap.lothar.com/trac>)
- pyOpenSSL (<https://launchpad.net/pyopenssl>)
- IPython (<http://ipython.scipy.org>)

In addition, the following dependencies are needed to run the demos described in this document.

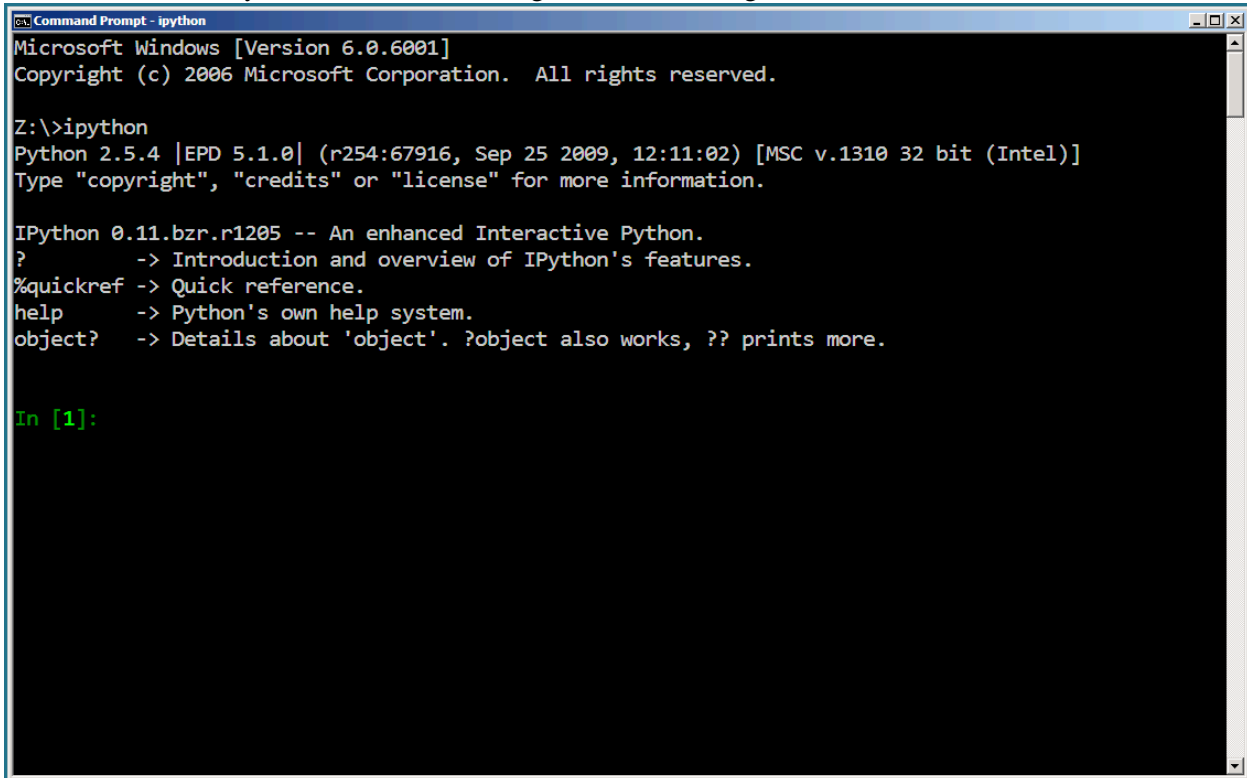
- NumPy and SciPy (<http://www.scipy.org>)
- wxPython (<http://www.wxpython.org>)
- Matplotlib (<http://matplotlib.sourceforge.net/>)

The easiest way of obtaining these dependencies is through the Enthought Python Distribution (EPD) (<http://www.enthought.com/products/epd.php>). EPD is produced by Enthought, Inc. and contains all of these packages and others in a single installer and is available free for academic users. While it is also possible to download and install each package individually, this is a tedious process. Thus, we highly recommend using EPD to install these packages on Windows.

Regardless of how you install the dependencies, here are the steps you will need to follow:

1. Install all of the packages listed above, either individually or using EPD on the head node, compute nodes and user workstations.
2. Make sure that `C:\Python25` and `C:\Python25\Scripts` are in the system `%PATH%` variable on each node.
3. Install the latest development version of IPython. This can be done by downloading the the development version from the IPython website (<http://ipython.scipy.org>) and following the installation instructions.

Further details about installing IPython or its dependencies can be found in the online IPython documentation (<http://ipython.scipy.org/moin/Documentation>) Once you are finished with the installation, you can try IPython out by opening a Windows Command Prompt and typing `ipython`. This will start IPython's interactive shell and you should see something like the following screenshot:



```
Command Prompt - ipython
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

Z:\>ipython
Python 2.5.4 |EPD 5.1.0| (r254:67916, Sep 25 2009, 12:11:02) [MSC v.1310 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.11.bzr.r1205 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

5.7.4 Starting an IPython cluster

To use IPython's parallel computing capabilities, you will need to start an IPython cluster. An IPython cluster consists of one controller and multiple engines:

IPython controller The IPython controller manages the engines and acts as a gateway between the engines and the client, which runs in the user's interactive IPython session. The controller is started using the **ipcontroller** command.

IPython engine IPython engines run a user's Python code in parallel on the compute nodes. Engines are starting using the **ipengine** command.

Once these processes are started, a user can run Python code interactively and in parallel on the engines from within the IPython shell using an appropriate client. This includes the ability to interact with, plot and visualize data from the engines.

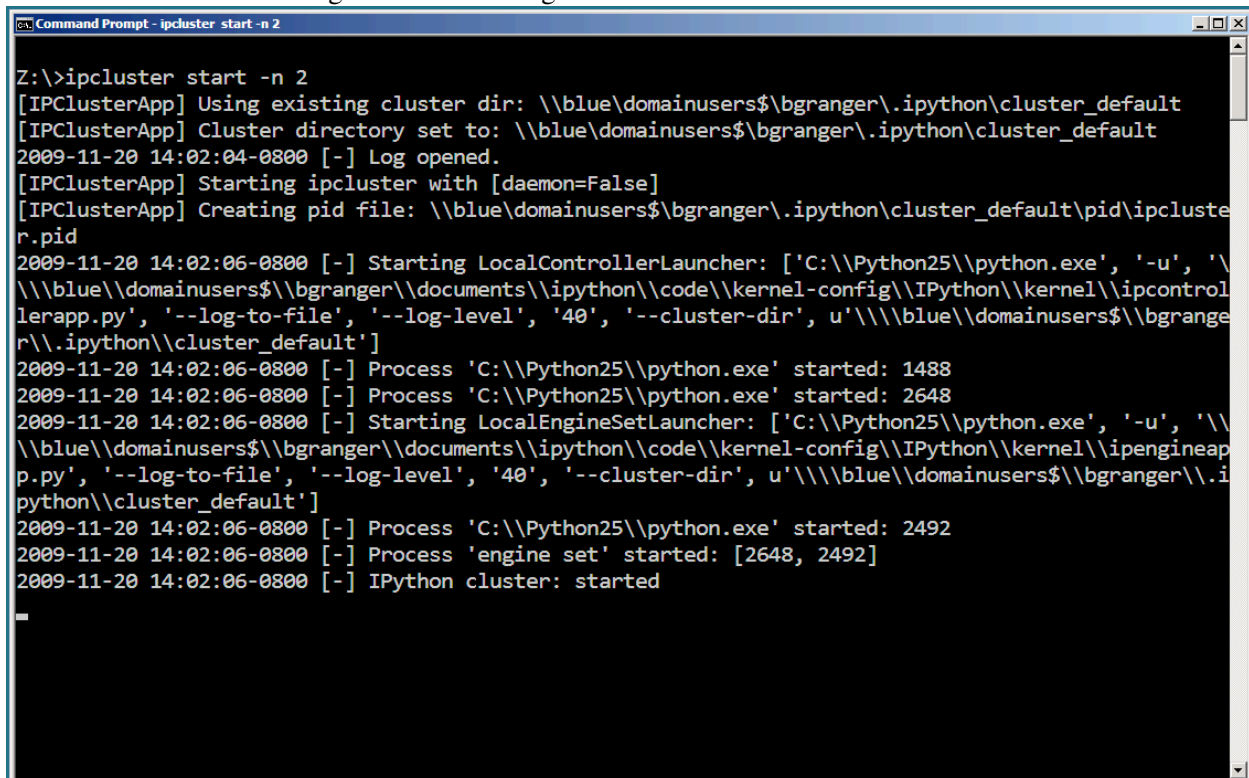
IPython has a command line program called **ipcluster** that automates all aspects of starting the controller and engines on the compute nodes. **ipcluster** has full support for the Windows HPC job scheduler, meaning that **ipcluster** can use this job scheduler to start the controller and engines. In our experience, the Windows HPC job scheduler is particularly well suited for interactive applications, such as IPython. Once **ipcluster** is configured properly, a user can start an IPython cluster from their local workstation almost instantly, without

having to log on to the head node (as is typically required by Unix based job schedulers). This enables a user to move seamlessly between serial and parallel computations.

In this section we show how to use **ipcluster** to start an IPython cluster using the Windows HPC Server 2008 job scheduler. To make sure that **ipcluster** is installed and working properly, you should first try to start an IPython cluster on your local host. To do this, open a Windows Command Prompt and type the following command:

```
ipcluster start -n 2
```

You should see a number of messages printed to the screen, ending with “IPython cluster: started”. The result should look something like the following screenshot:



```

Z:\>ipcluster start -n 2
[IPClusterApp] Using existing cluster dir: \\blue\domainusers$bgranger\ipython\cluster_default
[IPClusterApp] Cluster directory set to: \\blue\domainusers$bgranger\ipython\cluster_default
2009-11-20 14:02:04-0800 [-] Log opened.
[IPClusterApp] Starting ipcluster with [daemon=False]
[IPClusterApp] Creating pid file: \\blue\domainusers$bgranger\ipython\cluster_default\pid\ipcluster.pid
2009-11-20 14:02:06-0800 [-] Starting LocalControllerLauncher: ['C:\\Python25\\python.exe', '-u', '\\\\blue\\domainusers$bgranger\\documents\\ipython\\code\\kernel-config\\IPython\\kernel\\ipcontrollerapp.py', '--log-to-file', '--log-level', '40', '--cluster-dir', u'\\\\blue\\domainusers$bgranger\\ipython\\cluster_default']
2009-11-20 14:02:06-0800 [-] Process 'C:\\Python25\\python.exe' started: 1488
2009-11-20 14:02:06-0800 [-] Process 'C:\\Python25\\python.exe' started: 2648
2009-11-20 14:02:06-0800 [-] Starting LocalEngineSetLauncher: ['C:\\Python25\\python.exe', '-u', '\\\\blue\\domainusers$bgranger\\documents\\ipython\\code\\kernel-config\\IPython\\kernel\\ipengineapp.py', '--log-to-file', '--log-level', '40', '--cluster-dir', u'\\\\blue\\domainusers$bgranger\\ipython\\cluster_default']
2009-11-20 14:02:06-0800 [-] Process 'C:\\Python25\\python.exe' started: 2492
2009-11-20 14:02:06-0800 [-] Process 'engine set' started: [2648, 2492]
2009-11-20 14:02:06-0800 [-] IPython cluster: started

```

At this point, the controller and two engines are running on your local host. This configuration is useful for testing and for situations where you want to take advantage of multiple cores on your local computer.

Now that we have confirmed that **ipcluster** is working properly, we describe how to configure and run an IPython cluster on an actual compute cluster running Windows HPC Server 2008. Here is an outline of the needed steps:

1. Create a cluster profile using: `ipcluster create -p mycluster`
2. Edit configuration files in the directory `.ipython\cluster_mycluster`
3. Start the cluster using: `ipcluster start -p mycluster -n 32`

Creating a cluster profile

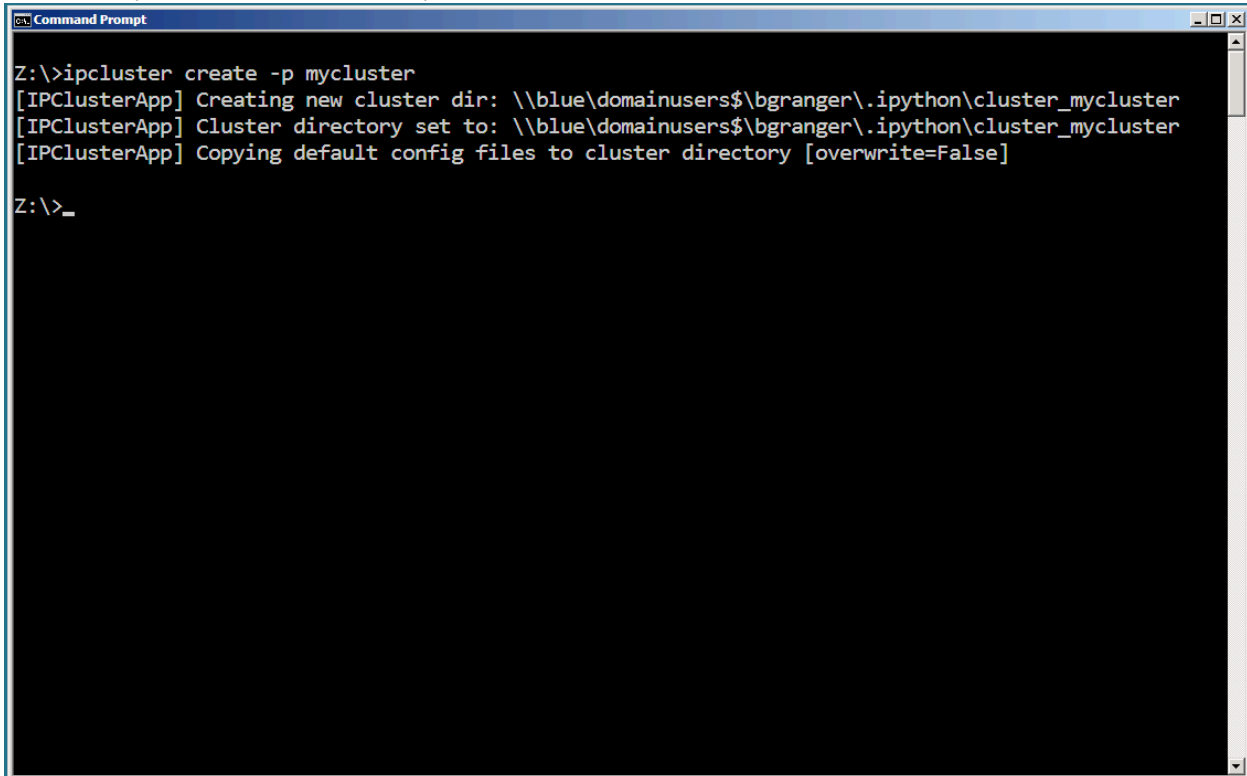
In most cases, you will have to create a cluster profile to use IPython on a cluster. A cluster profile is a name (like “mycluster”) that is associated with a particular cluster configuration. The profile name is used by **ipcluster** when working with the cluster.

Associated with each cluster profile is a cluster directory. This cluster directory is a specially named directory (typically located in the `.ipython` subdirectory of your home directory) that contains the configuration files for a particular cluster profile, as well as log files and security keys. The naming convention for cluster directories is: `cluster_<profile name>`. Thus, the cluster directory for a profile named “foo” would be `.ipython\cluster_foo`.

To create a new cluster profile (named “mycluster”) and the associated cluster directory, type the following command at the Windows Command Prompt:

```
ipcluster create -p mycluster
```

The output of this command is shown in the screenshot below. Notice how **ipcluster** prints out the location of the newly created cluster directory.



```
Command Prompt
Z:\>ipcluster create -p mycluster
[IPClusterApp] Creating new cluster dir: \\blue\domainusers$bgranger\.ipython\cluster_mycluster
[IPClusterApp] Cluster directory set to: \\blue\domainusers$bgranger\.ipython\cluster_mycluster
[IPClusterApp] Copying default config files to cluster directory [overwrite=False]
Z:\>_
```

Configuring a cluster profile

Next, you will need to configure the newly created cluster profile by editing the following configuration files in the cluster directory:

- `ipcluster_config.py`
- `ipcontroller_config.py`

- `ipengine_config.py`

When **ipcluster** is run, these configuration files are used to determine how the engines and controller will be started. In most cases, you will only have to set a few of the attributes in these files.

To configure **ipcluster** to use the Windows HPC job scheduler, you will need to edit the following attributes in the file `ipcluster_config.py`:

```
# Set these at the top of the file to tell ipcluster to use the
# Windows HPC job scheduler.
c.Global.controller_launcher = \
    'IPython.kernel.launcher.WindowsHPCControllerLauncher'
c.Global.engine_launcher = \
    'IPython.kernel.launcher.WindowsHPCEngineSetLauncher'

# Set these to the host name of the scheduler (head node) of your cluster.
c.WindowsHPCControllerLauncher.scheduler = 'HEADNODE'
c.WindowsHPCEngineSetLauncher.scheduler = 'HEADNODE'
```

There are a number of other configuration attributes that can be set, but in most cases these will be sufficient to get you started.

Warning: If any of your configuration attributes involve specifying the location of shared directories or files, you must make sure that you use UNC paths like `\\host\share`. It is also important that you specify these paths using raw Python strings: `r'\\host\share'` to make sure that the backslashes are properly escaped.

Starting the cluster profile

Once a cluster profile has been configured, starting an IPython cluster using the profile is simple:

```
ipcluster start -p mycluster -n 32
```

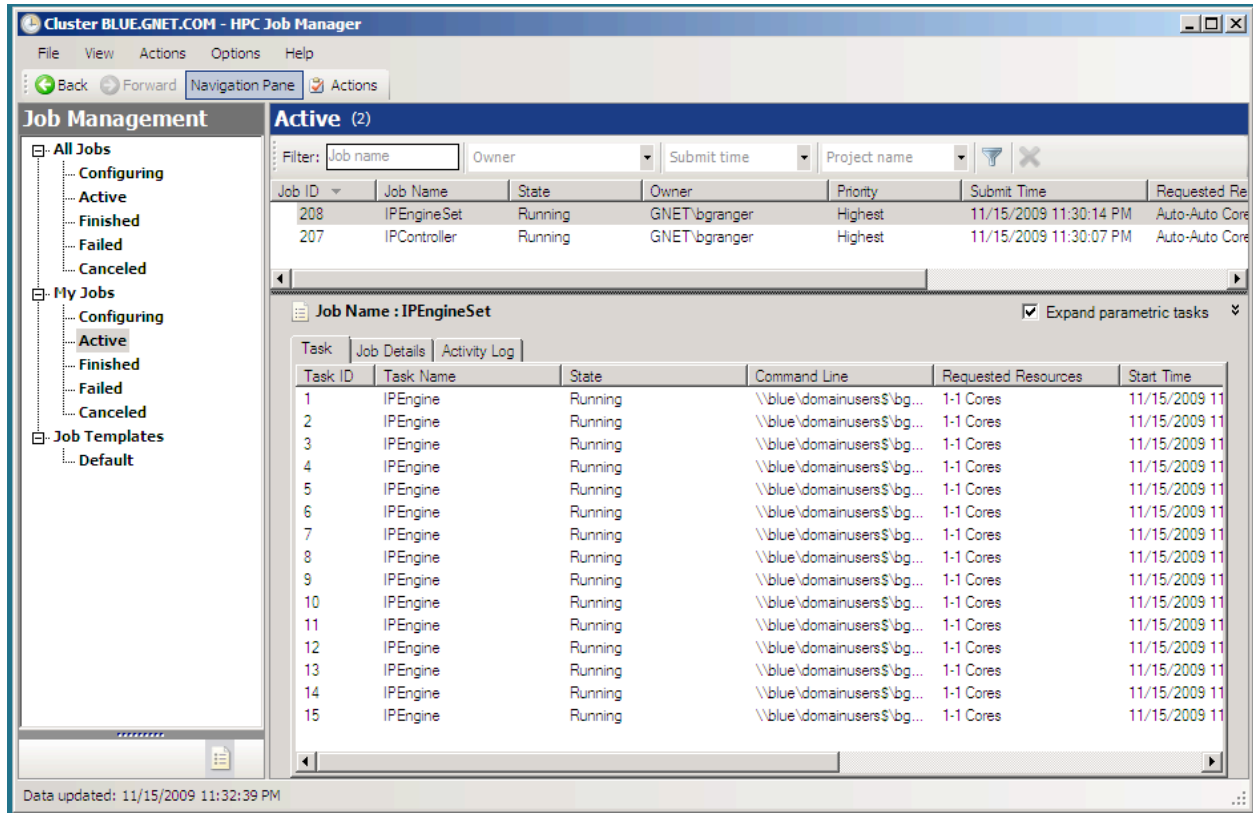
The `-n` option tells **ipcluster** how many engines to start (in this case 32). Stopping the cluster is as simple as typing Control-C.

Using the HPC Job Manager

When `ipcluster start` is run the first time, **ipcluster** creates two XML job description files in the cluster directory:

- `ipcontroller_job.xml`
- `ipengineset_job.xml`

Once these files have been created, they can be imported into the HPC Job Manager application. Then, the controller and engines for that profile can be started using the HPC Job Manager directly, without using **ipcluster**. However, anytime the cluster profile is re-configured, `ipcluster start` must be run again to regenerate the XML job description files. The following screenshot shows what the HPC Job Manager interface looks like with a running IPython cluster.



5.7.5 Performing a simple interactive parallel computation

Once you have started your IPython cluster, you can start to use it. To do this, open up a new Windows Command Prompt and start up IPython's interactive shell by typing:

```
ipython
```

Then you can create a `MultiEngineClient` instance for your profile and use the resulting instance to do a simple interactive parallel computation. In the code and screenshot that follows, we take a simple Python function and apply it to each element of an array of integers in parallel using the `MultiEngineClient.map()` method:

```
In [1]: from IPython.kernel.client import *
```

```
In [2]: mec = MultiEngineClient(profile='mycluster')
```

```
In [3]: mec.get_ids()
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 67, 8, 9, 10, 11, 12, 13, 14]
```

```
In [4]: def f(x):
...:     return x**10
```

```
In [5]: mec.map(f, range(15)) # f is applied in parallel
```

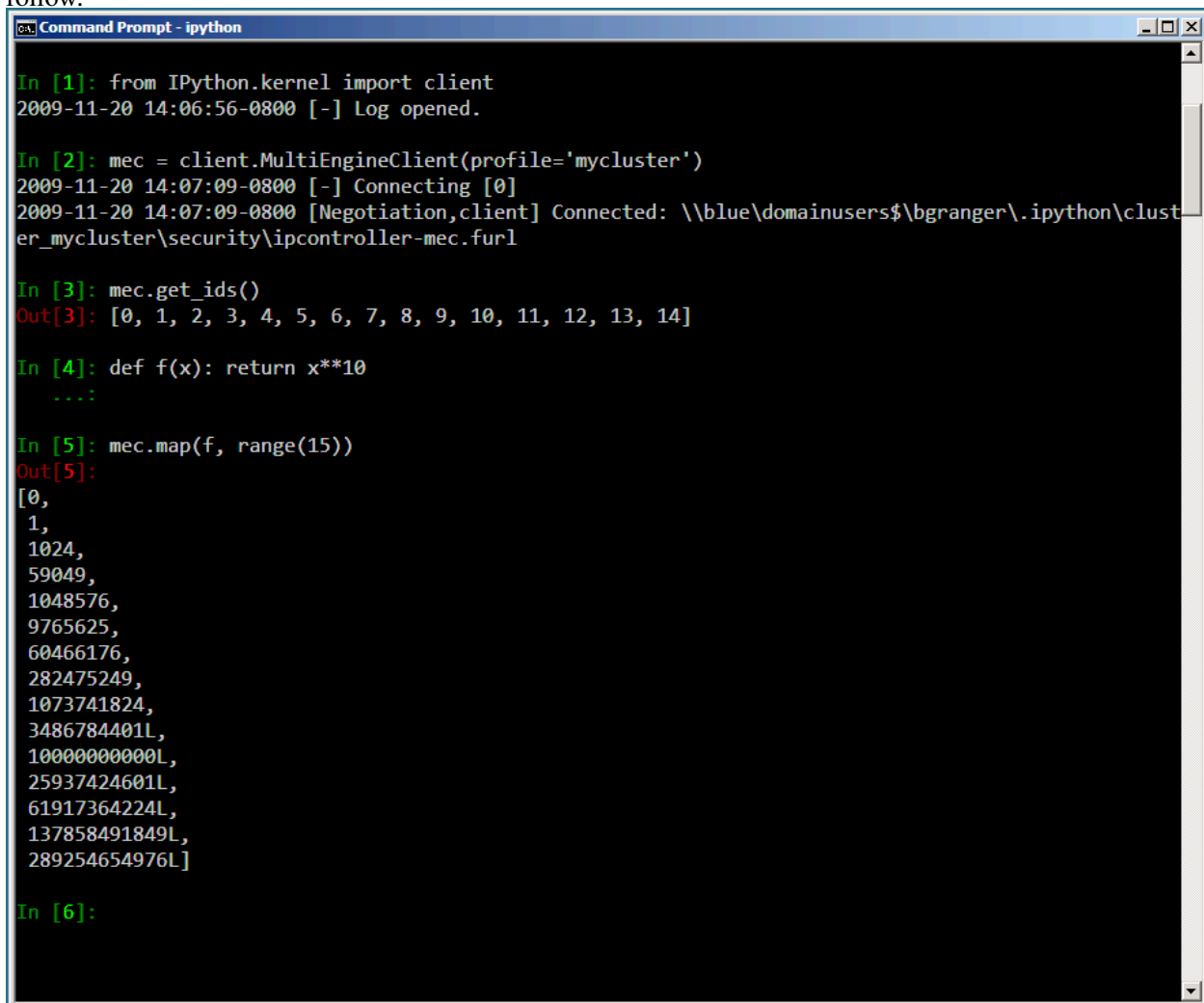
```
Out[5]:
[0,
 1,
```

```

1024,
59049,
1048576,
9765625,
60466176,
282475249,
1073741824,
3486784401L,
10000000000L,
25937424601L,
61917364224L,
137858491849L,
289254654976L]

```

The `map()` method has the same signature as Python's builtin `map()` function, but runs the calculation in parallel. More involved examples of using `MultiEngineClient` are provided in the examples that follow.



```

C:\> Command Prompt - ipython

In [1]: from IPython.kernel import client
2009-11-20 14:06:56-0800 [-] Log opened.

In [2]: mec = client.MultiEngineClient(profile='mycluster')
2009-11-20 14:07:09-0800 [-] Connecting [0]
2009-11-20 14:07:09-0800 [Negotiation,client] Connected: \\blue\domainusers$\bgranger\.ipython\cluster_mycluster\security\ipcontroller-mec.furl

In [3]: mec.get_ids()
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

In [4]: def f(x): return x**10
...:

In [5]: mec.map(f, range(15))
Out[5]:
[0,
1,
1024,
59049,
1048576,
9765625,
60466176,
282475249,
1073741824,
3486784401L,
10000000000L,
25937424601L,
61917364224L,
137858491849L,
289254654976L]

In [6]:

```

5.8 Parallel examples

In this section we describe two more involved examples of using an IPython cluster to perform a parallel computation. In these examples, we will be using IPython’s “pylab” mode, which enables interactive plotting using the Matplotlib package. IPython can be started in this mode by typing:

```
ipython -p pylab
```

at the system command line. If this prints an error message, you will need to install the default profiles from within IPython by doing,

```
In [1]: %install_profiles
```

and then restarting IPython.

5.8.1 150 million digits of pi

In this example we would like to study the distribution of digits in the number pi (in base 10). While it is not known if pi is a normal number (a number is normal in base 10 if 0-9 occur with equal likelihood) numerical investigations suggest that it is. We will begin with a serial calculation on 10,000 digits of pi and then perform a parallel calculation involving 150 million digits.

In both the serial and parallel calculation we will be using functions defined in the `pidigits.py` file, which is available in the `docs/examples/kernel` directory of the IPython source distribution. These functions provide basic facilities for working with the digits of pi and can be loaded into IPython by putting `pidigits.py` in your current working directory and then doing:

```
In [1]: run pidigits.py
```

Serial calculation

For the serial calculation, we will use SymPy (<http://www.sympy.org>) to calculate 10,000 digits of pi and then look at the frequencies of the digits 0-9. Out of 10,000 digits, we expect each digit to occur 1,000 times. While SymPy is capable of calculating many more digits of pi, our purpose here is to set the stage for the much larger parallel calculation.

In this example, we use two functions from `pidigits.py`: `one_digit_freqs()` (which calculates how many times each digit occurs) and `plot_one_digit_freqs()` (which uses Matplotlib to plot the result). Here is an interactive IPython session that uses these functions with SymPy:

```
In [7]: import sympy
```

```
In [8]: pi = sympy.pi.evalf(40)
```

```
In [9]: pi
```

```
Out[9]: 3.141592653589793238462643383279502884197
```

```
In [10]: pi = sympy.pi.evalf(10000)
```

```
In [11]: digits = (d for d in str(pi)[2:]) # create a sequence of digits
```



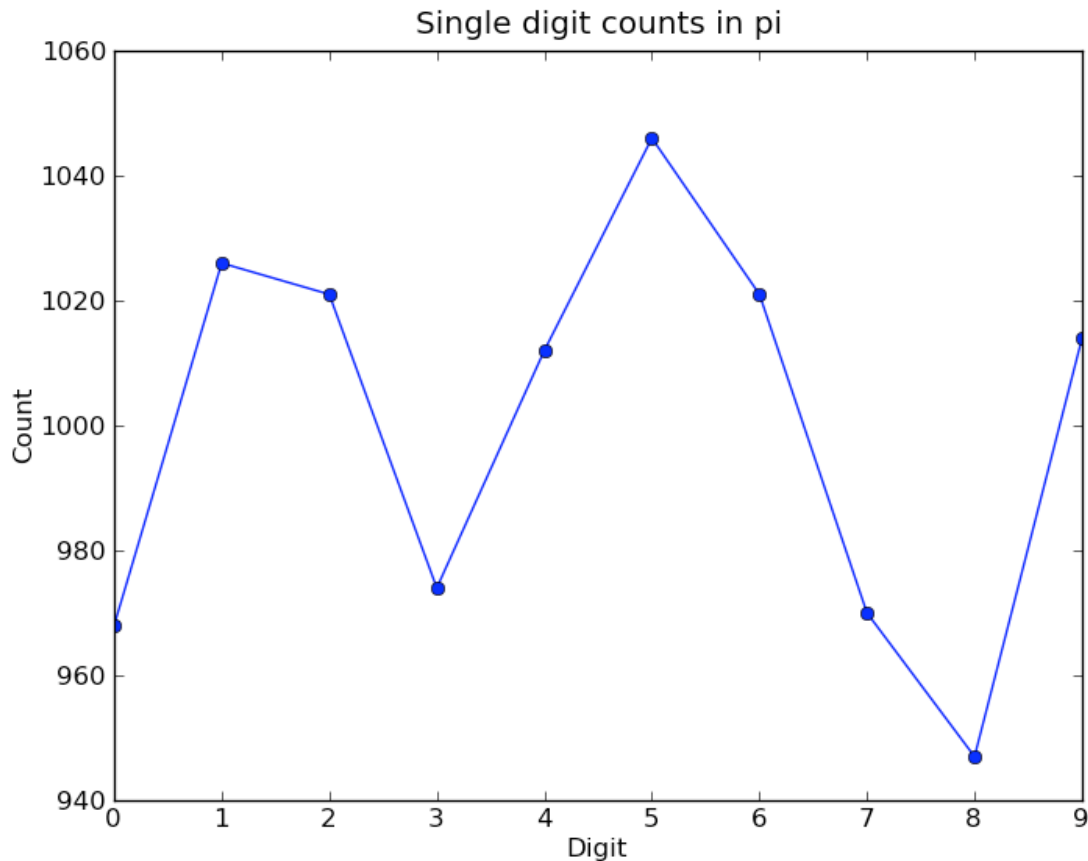
```
In [12]: run pidigits.py # load one_digit_freqs/plot_one_digit_freqs
```

```
In [13]: freqs = one_digit_freqs(digits)
```

```
In [14]: plot_one_digit_freqs(freqs)
```

```
Out[14]: [<matplotlib.lines.Line2D object at 0x18a55290>]
```

The resulting plot of the single digit counts shows that each digit occurs approximately 1,000 times, but that with only 10,000 digits the statistical fluctuations are still rather large:



It is clear that to reduce the relative fluctuations in the counts, we need to look at many more digits of pi. That brings us to the parallel calculation.

Parallel calculation

Calculating many digits of pi is a challenging computational problem in itself. Because we want to focus on the distribution of digits in this example, we will use pre-computed digit of pi from the website of Professor Yasumasa Kanada at the University of Tokyo (<http://www.super-computing.org>). These digits come in a set of text files (<ftp://pi.super-computing.org/.2/pi200m/>) that each have 10 million digits of pi.

For the parallel calculation, we have copied these files to the local hard drives of the compute nodes. A total of 15 of these files will be used, for a total of 150 million digits of pi. To make things a little more interesting

we will calculate the frequencies of all 2 digits sequences (00-99) and then plot the result using a 2D matrix in Matplotlib.

The overall idea of the calculation is simple: each IPython engine will compute the two digit counts for the digits in a single file. Then in a final step the counts from each engine will be added up. To perform this calculation, we will need two top-level functions from `pidigits.py`:

```
def compute_two_digit_freqs(filename):
    """
    Read digits of pi from a file and compute the 2 digit frequencies.
    """
    d = txt_file_to_digits(filename)
    freqs = two_digit_freqs(d)
    return freqs

def reduce_freqs(freqlist):
    """
    Add up a list of freq counts to get the total counts.
    """
    allfreqs = np.zeros_like(freqlist[0])
    for f in freqlist:
        allfreqs += f
    return allfreqs
```

We will also use the `plot_two_digit_freqs()` function to plot the results. The code to run this calculation in parallel is contained in `docs/examples/kernel/parallempi.py`. This code can be run in parallel using IPython by following these steps:

1. Copy the text files with the digits of pi (<ftp://pi.super-computing.org/.2/pi200m/>) to the working directory of the engines on the compute nodes.
2. Use **ipcluster** to start 15 engines. We used an 8 core (2 quad core CPUs) cluster with hyperthreading enabled which makes the 8 cores look like 16 (1 controller + 15 engines) in the OS. However, the maximum speedup we can observe is still only 8x.
3. With the file `parallempi.py` in your current working directory, open up IPython in pylab mode and type `run parallempi.py`.

When run on our 8 core cluster, we observe a speedup of 7.7x. This is slightly less than linear scaling (8x) because the controller is also running on one of the cores.

To emphasize the interactive nature of IPython, we now show how the calculation can also be run by simply typing the commands from `parallempi.py` interactively into IPython:

```
In [1]: from IPython.kernel import client
2009-11-19 11:32:38-0800 [-] Log opened.

# The MultiEngineClient allows us to use the engines interactively.
# We simply pass MultiEngineClient the name of the cluster profile we
# are using.
In [2]: mec = client.MultiEngineClient(profile='mycluster')
2009-11-19 11:32:44-0800 [-] Connecting [0]
2009-11-19 11:32:44-0800 [Negotiation,client] Connected: ./ipcontroller-mec.furl
```

```

In [3]: mec.get_ids()
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

In [4]: run pidigits.py

In [5]: filestring = 'pi200m-ascii-%(i)02dof20.txt'

# Create the list of files to process.
In [6]: files = [filestring % {'i':i} for i in range(1,16)]

In [7]: files
Out[7]:
['pi200m-ascii-01of20.txt',
 'pi200m-ascii-02of20.txt',
 'pi200m-ascii-03of20.txt',
 'pi200m-ascii-04of20.txt',
 'pi200m-ascii-05of20.txt',
 'pi200m-ascii-06of20.txt',
 'pi200m-ascii-07of20.txt',
 'pi200m-ascii-08of20.txt',
 'pi200m-ascii-09of20.txt',
 'pi200m-ascii-10of20.txt',
 'pi200m-ascii-11of20.txt',
 'pi200m-ascii-12of20.txt',
 'pi200m-ascii-13of20.txt',
 'pi200m-ascii-14of20.txt',
 'pi200m-ascii-15of20.txt']

# This is the parallel calculation using the MultiEngineClient.map method
# which applies compute_two_digit_freqs to each file in files in parallel.
In [8]: freqs_all = mec.map(compute_two_digit_freqs, files)

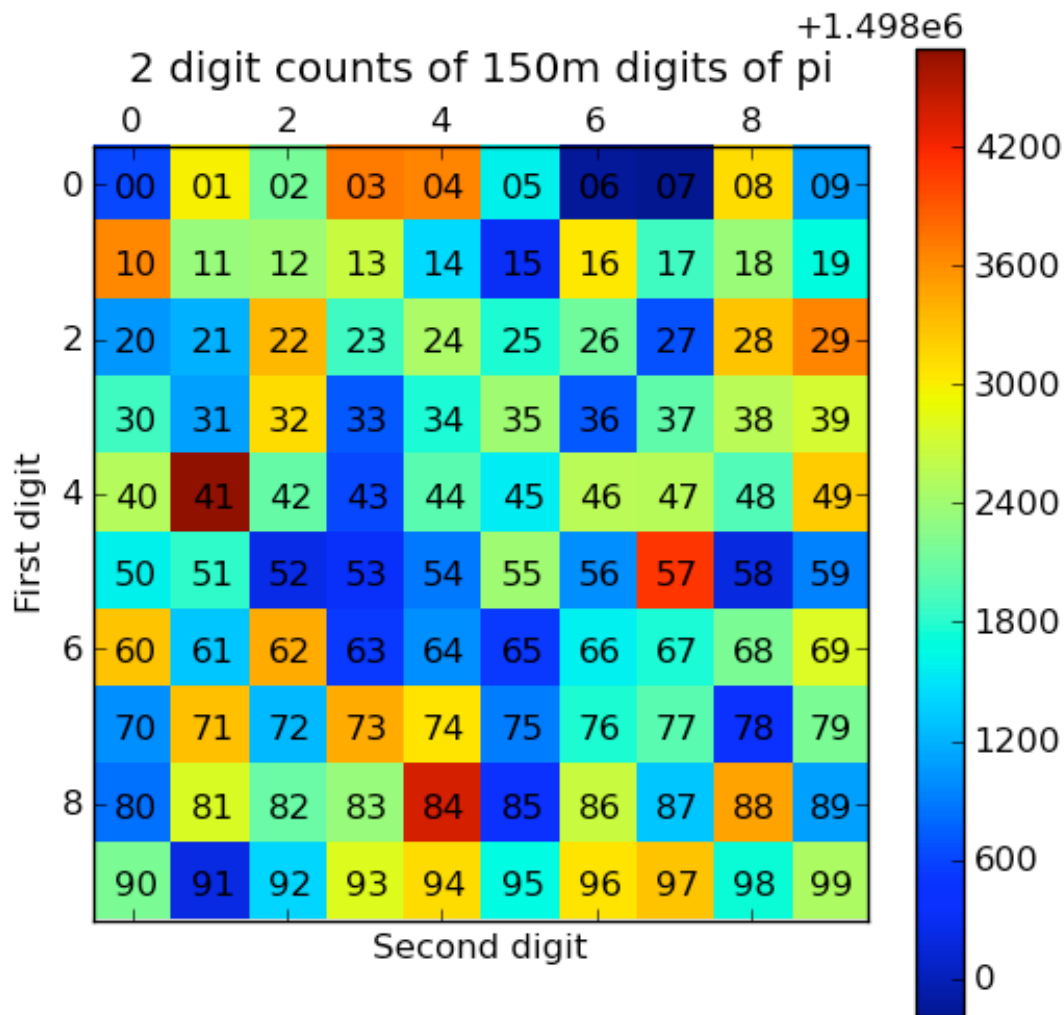
# Add up the frequencies from each engine.
In [8]: freqs = reduce_freqs(freqs_all)

In [9]: plot_two_digit_freqs(freqs)
Out[9]: <matplotlib.image.AxesImage object at 0x18beb110>

In [10]: plt.title('2 digit counts of 150m digits of pi')
Out[10]: <matplotlib.text.Text object at 0x18d1f9b0>

```

The resulting plot generated by Matplotlib is shown below. The colors indicate which two digit sequences are more (red) or less (blue) likely to occur in the first 150 million digits of pi. We clearly see that the sequence “41” is most likely and that “06” and “07” are least likely. Further analysis would show that the relative size of the statistical fluctuations have decreased compared to the 10,000 digit calculation.



5.8.2 Parallel options pricing

An option is a financial contract that gives the buyer of the contract the right to buy (a “call”) or sell (a “put”) a secondary asset (a stock for example) at a particular date in the future (the expiration date) for a pre-agreed upon price (the strike price). For this right, the buyer pays the seller a premium (the option price). There are a wide variety of flavors of options (American, European, Asian, etc.) that are useful for different purposes: hedging against risk, speculation, etc.

Much of modern finance is driven by the need to price these contracts accurately based on what is known about the properties (such as volatility) of the underlying asset. One method of pricing options is to use a Monte Carlo simulation of the underlying asset price. In this example we use this approach to price both European and Asian (path dependent) options for various strike prices and volatilities.

The code for this example can be found in the docs/examples/kernel directory of the IPython source. The function `price_options()` in `mcpricer.py` implements the basic Monte Carlo pricing algorithm using the NumPy package and is shown here:

```
import numpy as np
from math import *
```



```
def price_options(S=100.0, K=100.0, sigma=0.25, r=0.05, days=260, paths=10000):
    """
    Price European and Asian options using a Monte Carlo method.

    Parameters
    -----
    S : float
        The initial price of the stock.
    K : float
        The strike price of the option.
    sigma : float
        The volatility of the stock.
    r : float
        The risk free interest rate.
    days : int
        The number of days until the option expires.
    paths : int
        The number of Monte Carlo paths used to price the option.

    Returns
    -----
    A tuple of (E. call, E. put, A. call, A. put) option prices.
    """
    h = 1.0/days
    const1 = exp((r-0.5*sigma**2)*h)
    const2 = sigma*sqrt(h)
    stock_price = S*np.ones(paths, dtype='float64')
    stock_price_sum = np.zeros(paths, dtype='float64')
    for j in range(days):
        growth_factor = const1*np.exp(const2*np.random.standard_normal(paths))
        stock_price = stock_price*growth_factor
        stock_price_sum = stock_price_sum + stock_price
    stock_price_avg = stock_price_sum/days
    zeros = np.zeros(paths, dtype='float64')
    r_factor = exp(-r*h*days)
    euro_put = r_factor*np.mean(np.maximum(zeros, K-stock_price))
    asian_put = r_factor*np.mean(np.maximum(zeros, K-stock_price_avg))
    euro_call = r_factor*np.mean(np.maximum(zeros, stock_price-K))
    asian_call = r_factor*np.mean(np.maximum(zeros, stock_price_avg-K))
    return (euro_call, euro_put, asian_call, asian_put)
```

To run this code in parallel, we will use IPython's `TaskClient` class, which distributes work to the engines using dynamic load balancing. This client can be used along side the `MultiEngineClient` class shown in the previous example. The parallel calculation using `TaskClient` can be found in the file `mcpricer.py`. The code in this file creates a `TaskClient` instance and then submits a set of tasks

using `TaskClient.run()` that calculate the option prices for different volatilities and strike prices. The results are then plotted as a 2D contour plot using Matplotlib.

```
#!/usr/bin/env python
"""Run a Monte-Carlo options pricer in parallel."""

#-----
# Imports
#-----

import sys
import time
from IPython.kernel import client
import numpy as np
from mcpricer import price_options
from matplotlib import pyplot as plt

#-----
# Setup parameters for the run
#-----

def ask_question(text, the_type, default):
    s = '%s [%r]: ' % (text, the_type(default))
    result = raw_input(s)
    if result:
        return the_type(result)
    else:
        return the_type(default)

cluster_profile = ask_question("Cluster profile", str, "default")
price = ask_question("Initial price", float, 100.0)
rate = ask_question("Interest rate", float, 0.05)
days = ask_question("Days to expiration", int, 260)
paths = ask_question("Number of MC paths", int, 10000)
n_strikes = ask_question("Number of strike values", int, 5)
min_strike = ask_question("Min strike price", float, 90.0)
max_strike = ask_question("Max strike price", float, 110.0)
n_sigmas = ask_question("Number of volatility values", int, 5)
min_sigma = ask_question("Min volatility", float, 0.1)
max_sigma = ask_question("Max volatility", float, 0.4)

strike_vals = np.linspace(min_strike, max_strike, n_strikes)
sigma_vals = np.linspace(min_sigma, max_sigma, n_sigmas)

#-----
# Setup for parallel calculation
#-----

# The MultiEngineClient is used to setup the calculation and works with all
# engine.
mec = client.MultiEngineClient(profile=cluster_profile)

# The TaskClient is an interface to the engines that provides dynamic load
# balancing at the expense of not knowing which engine will execute the code.
```

```

tc = client.TaskClient(profile=cluster_profile)

# Initialize the common code on the engines. This Python module has the
# price_options function that prices the options.
mec.run('mcpricer.py')

#-----
# Perform parallel calculation
#-----

print "Running parallel calculation over strike prices and volatilities..."
print "Strike prices: ", strike_vals
print "Volatilities: ", sigma_vals
sys.stdout.flush()

# Submit tasks to the TaskClient for each (strike, sigma) pair as a MapTask.
t1 = time.time()
taskids = []
for strike in strike_vals:
    for sigma in sigma_vals:
        t = client.MapTask(
            price_options,
            args=(price, strike, sigma, rate, days, paths)
        )
        taskids.append(tc.run(t))

print "Submitted tasks: ", len(taskids)
sys.stdout.flush()

# Block until all tasks are completed.
tc.barrier(taskids)
t2 = time.time()
t = t2-t1

print "Parallel calculation completed, time = %s s" % t
print "Collecting results..."

# Get the results using TaskClient.get_task_result.
results = [tc.get_task_result(tid) for tid in taskids]

# Assemble the result into a structured NumPy array.
prices = np.empty(n_strikes*n_sigmas,
    dtype=[('ecall', float), ('eput', float), ('acall', float), ('aput', float)]
)

for i, price_tuple in enumerate(results):
    prices[i] = price_tuple

prices.shape = (n_strikes, n_sigmas)
strike_mesh, sigma_mesh = np.meshgrid(strike_vals, sigma_vals)

print "Results are available: strike_mesh, sigma_mesh, prices"
print "To plot results type 'plot_options(sigma_mesh, strike_mesh, prices)'"

```

```
#-----  
# Utilities  
#-----  
  
def plot_options(sigma_mesh, strike_mesh, prices):  
    """  
    Make a contour plot of the option price in (sigma, strike) space.  
    """  
    plt.figure(1)  
  
    plt.subplot(221)  
    plt.contourf(sigma_mesh, strike_mesh, prices['ecall'])  
    plt.axis('tight')  
    plt.colorbar()  
    plt.title('European Call')  
    plt.ylabel("Strike Price")  
  
    plt.subplot(222)  
    plt.contourf(sigma_mesh, strike_mesh, prices['acall'])  
    plt.axis('tight')  
    plt.colorbar()  
    plt.title("Asian Call")  
  
    plt.subplot(223)  
    plt.contourf(sigma_mesh, strike_mesh, prices['eput'])  
    plt.axis('tight')  
    plt.colorbar()  
    plt.title("European Put")  
    plt.xlabel("Volatility")  
    plt.ylabel("Strike Price")  
  
    plt.subplot(224)  
    plt.contourf(sigma_mesh, strike_mesh, prices['aput'])  
    plt.axis('tight')  
    plt.colorbar()  
    plt.title("Asian Put")  
    plt.xlabel("Volatility")
```

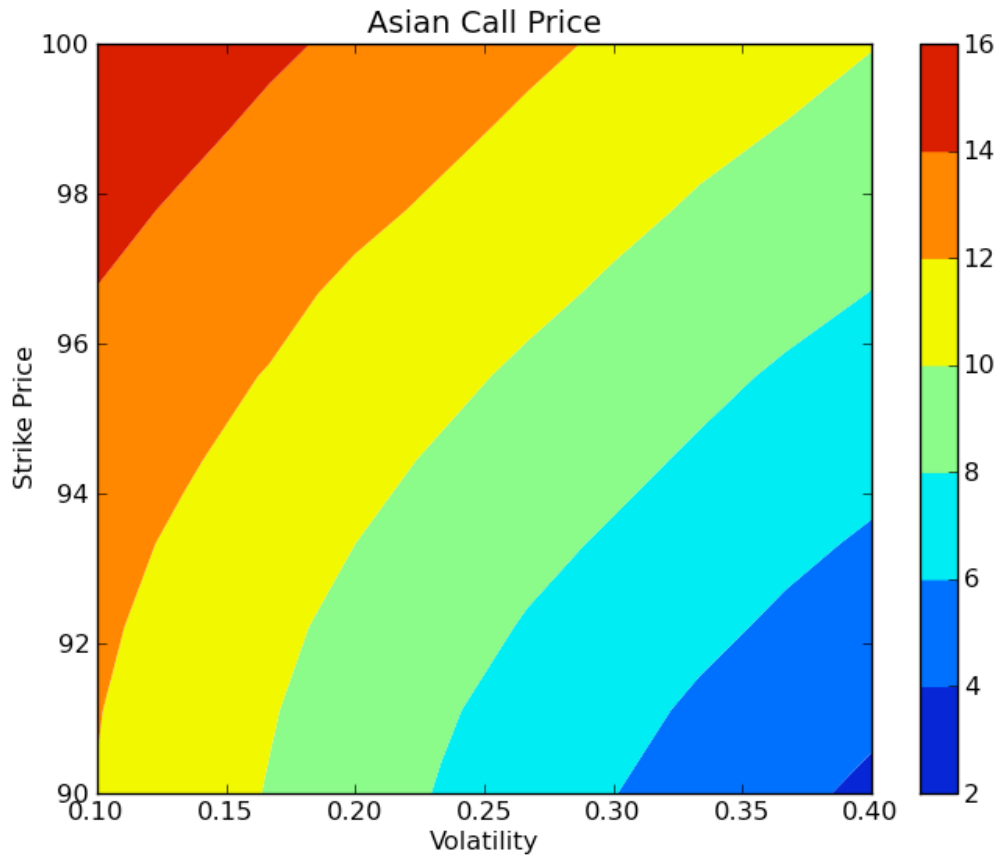
To use this code, start an IPython cluster using **ipcluster**, open IPython in the pylab mode with the file `mcdriver.py` in your current working directory and then type:

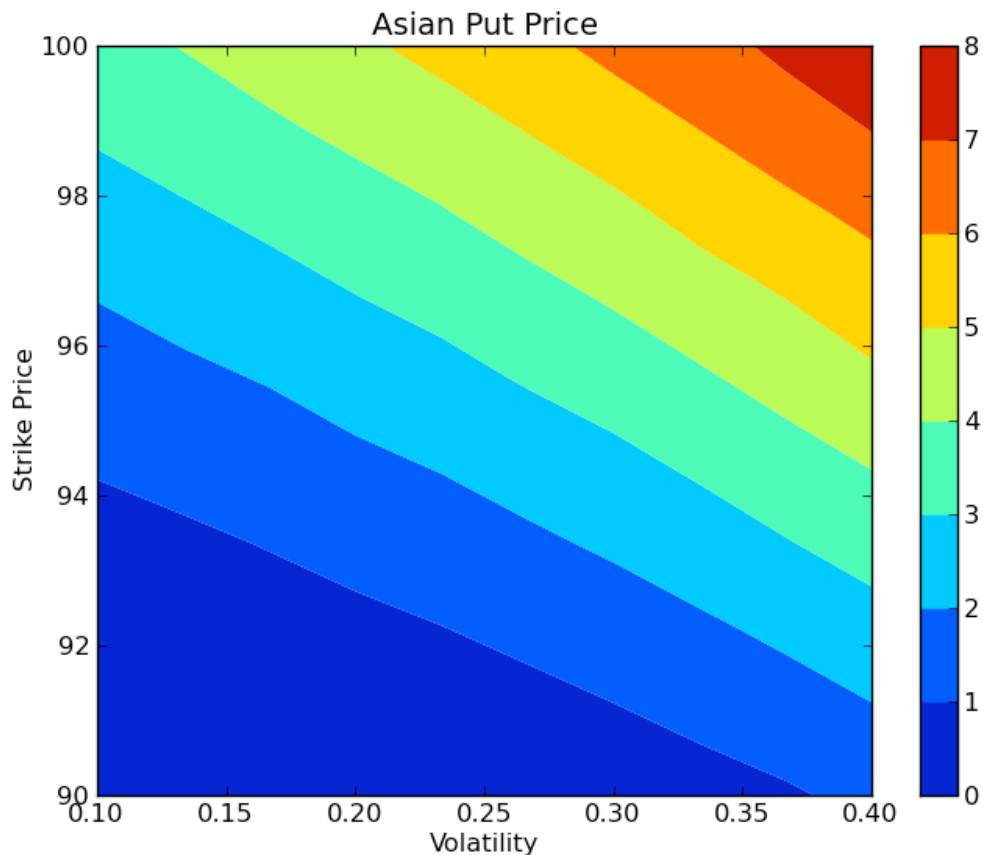
```
In [7]: run mcdriver.py  
Submitted tasks: [0, 1, 2, ...]
```

Once all the tasks have finished, the results can be plotted using the `plot_options()` function. Here we make contour plots of the Asian call and Asian put options as function of the volatility and strike price:

```
In [8]: plot_options(sigma_vals, K_vals, prices['acall'])  
  
In [9]: plt.figure()  
Out[9]: <matplotlib.figure.Figure object at 0x18c178d0>  
  
In [10]: plot_options(sigma_vals, K_vals, prices['aput'])
```


These results are shown in the two figures below. On a 8 core cluster the entire calculation (10 strike prices, 10 volatilities, 100,000 paths for each) took 30 seconds in parallel, giving a speedup of 7.7x, which is comparable to the speedup observed in our previous example.





5.8.3 Conclusion

To conclude these examples, we summarize the key features of IPython's parallel architecture that have been demonstrated:

- Serial code can be parallelized often with only a few extra lines of code. We have used the `MultiEngineClient` and `TaskClient` classes for this purpose.
- The resulting parallel code can be run without ever leaving the IPython's interactive shell.
- Any data computed in parallel can be explored interactively through visualization or further numerical calculations.
- We have run these examples on a cluster running Windows HPC Server 2008. IPython's built in support for the Windows HPC job scheduler makes it easy to get started with IPython's parallel capabilities.

CONFIGURATION AND CUSTOMIZATION

6.1 Overview of the IPython configuration system

This section describes the IPython configuration system. Starting with version 0.11, IPython has a completely new configuration system that is quite different from the older `ipythonrc` or `ipy_user_conf.py` approaches. The new configuration system was designed from scratch to address the particular configuration needs of IPython. While there are many other excellent configuration systems out there, we found that none of them met our requirements.

Warning: If you are upgrading to version 0.11 of IPython, you will need to migrate your old `ipythonrc` or `ipy_user_conf.py` configuration files to the new system. Read on for information on how to do this.

The discussion that follows is focused on teaching user's how to configure IPython to their liking. Developer's who want to know more about how they can enable their objects to take advantage of the configuration system should consult our *developer guide*

6.1.1 The main concepts

There are a number of abstractions that the IPython configuration system uses. Each of these abstractions is represented by a Python class.

Configuration object: **Config** A configuration object is a simple dictionary-like class that holds configuration attributes and sub-configuration objects. These classes support dotted attribute style access (`Foo.bar`) in addition to the regular dictionary style access (`Foo['bar']`). Configuration objects are smart. They know how to merge themselves with other configuration objects and they automatically create sub-configuration objects.

Application: **Application** An application is a process that does a specific job. The most obvious application is the **ipython** command line program. Each application reads a *single* configuration file and command line options and then produces a master configuration object for the application. This configuration object is then passed to the configurable objects that the application creates. These con-

figurable objects implement the actual logic of the application and know how to configure themselves given the configuration object.

Component: Configurable A configurable is a regular Python class that serves as a base class for all main classes in an application. The `Configurable` base class is lightweight and only does one things.

This `Configurable` is a subclass of `HasTraits` that knows how to configure itself. Class level traits with the metadata `config=True` become values that can be configured from the command line and configuration files.

Developers create `Configurable` subclasses that implement all of the logic in the application. Each of these subclasses has its own configuration information that controls how instances are created.

Having described these main concepts, we can now state the main idea in our configuration system: “*configuration*” allows the default values of class attributes to be controlled on a class by class basis. Thus all instances of a given class are configured in the same way. Furthermore, if two instances need to be configured differently, they need to be instances of two different classes. While this model may seem a bit restrictive, we have found that it expresses most things that need to be configured extremely well. However, it is possible to create two instances of the same class that have different trait values. This is done by overriding the configuration.

Now, we show what our configuration objects and files look like.

6.1.2 Configuration objects and files

A configuration file is simply a pure Python file that sets the attributes of a global, pre-created configuration object. This configuration object is a `Config` instance. While in a configuration file, to get a reference to this object, simply call the `get_config()` function. We inject this function into the global namespace that the configuration file is executed in.

Here is an example of a super simple configuration file that does nothing:

```
c = get_config()
```

Once you get a reference to the configuration object, you simply set attributes on it. All you have to know is:

- The name of each attribute.
- The type of each attribute.

The answers to these two questions are provided by the various `Configurable` subclasses that an application uses. Let’s look at how this would work for a simple component subclass:

```
# Sample component that can be configured.
from IPython.config.configurable import Configurable
from IPython.utils.traitlets import Int, Float, Str, Bool

class MyClass(Configurable):
    name = Str('defaultname', config=True)
    ranking = Int(0, config=True)
```

```
value = Float(99.0)
# The rest of the class implementation would go here..
```

In this example, we see that `MyClass` has three attributes, two of whom (`name`, `ranking`) can be configured. All of the attributes are given types and default values. If a `MyClass` is instantiated, but not configured, these default values will be used. But let's see how to configure this class in a configuration file:

```
# Sample config file
c = get_config()

c.MyClass.name = 'coolname'
c.MyClass.ranking = 10
```

After this configuration file is loaded, the values set in it will override the class defaults anytime a `MyClass` is created. Furthermore, these attributes will be type checked and validated anytime they are set. This type checking is handled by the `IPython.utils.traitlets` module, which provides the `Str`, `Int` and `Float` types. In addition to these `traitlets`, the `IPython.utils.traitlets` provides `traitlets` for a number of other types.

Note: Underneath the hood, the `Configurable` base class is a subclass of `IPython.utils.traitlets.HasTraits`. The `IPython.utils.traitlets` module is a lightweight version of `enthought.traits`. Our implementation is a pure Python subset (mostly API compatible) of `enthought.traits` that does not have any of the automatic GUI generation capabilities. Our plan is to achieve 100% API compatibility to enable the actual `enthought.traits` to eventually be used instead. Currently, we cannot use `enthought.traits` as we are committed to the core of IPython being pure Python.

It should be very clear at this point what the naming convention is for configuration attributes:

```
c.ClassName.attribute_name = attribute_value
```

Here, `ClassName` is the name of the class whose configuration attribute you want to set, `attribute_name` is the name of the attribute you want to set and `attribute_value` the the value you want it to have. The `ClassName` attribute of `c` is not the actual class, but instead is another `Config` instance.

Note: The careful reader may wonder how the `ClassName` (`MyClass` in the above example) attribute of the configuration object `c` gets created. These attributes are created on the fly by the `Config` instance, using a simple naming convention. Any attribute of a `Config` instance whose name begins with an uppercase character is assumed to be a sub-configuration and a new empty `Config` instance is dynamically created for that attribute. This allows deeply hierarchical information created easily (`c.Foo.Bar.value`) on the fly.

6.1.3 Configuration files inheritance

Let's say you want to have different configuration files for various purposes. Our configuration system makes it easy for one configuration file to inherit the information in another configuration file. The

`load_subconfig()` command can be used in a configuration file for this purpose. Here is a simple example that loads all of the values from the file `base_config.py`:

```
# base_config.py
c = get_config()
c.MyClass.name = 'coolname'
c.MyClass.ranking = 100
```

into the configuration file `main_config.py`:

```
# main_config.py
c = get_config()

# Load everything from base_config.py
load_subconfig('base_config.py')

# Now override one of the values
c.MyClass.name = 'bettername'
```

In a situation like this the `load_subconfig()` makes sure that the search path for sub-configuration files is inherited from that of the parent. Thus, you can typically put the two in the same directory and everything will just work.

6.1.4 Class based configuration inheritance

There is another aspect of configuration where inheritance comes into play. Sometimes, your classes will have an inheritance hierarchy that you want to be reflected in the configuration system. Here is a simple example:

```
from IPython.config.configurable import Configurable
from IPython.utils.traitlets import Int, Float, Str, Bool

class Foo(Configurable):
    name = Str('fooname', config=True)
    value = Float(100.0, config=True)

class Bar(Foo):
    name = Str('barname', config=True)
    othervalue = Int(0, config=True)
```

Now, we can create a configuration file to configure instances of `Foo` and `Bar`:

```
# config file
c = get_config()

c.Foo.name = 'bestname'
c.Bar.othervalue = 10
```

This class hierarchy and configuration file accomplishes the following:

- The default value for `Foo.name` and `Bar.name` will be 'bestname'. Because `Bar` is a `Foo` subclass it also picks up the configuration information for `Foo`.

- The default value for `Foo.value` and `Bar.value` will be `100.0`, which is the value specified as the class default.
- The default value for `Bar.othervalue` will be `10` as set in the configuration file. Because `Foo` is the parent of `Bar` it doesn't know anything about the `othervalue` attribute.

6.1.5 Configuration file location

So where should you put your configuration files? By default, all IPython applications look in the so called “IPython directory”. The location of this directory is determined by the following algorithm:

- If the `--ipython-dir` command line flag is given, its value is used.
- If not, the value returned by `IPython.utils.path.get_ipython_dir()` is used. This function will first look at the `IPYTHON_DIR` environment variable and then default to the directory `$HOME/.ipython`.

For most users, the default value will simply be something like `$HOME/.ipython`.

Once the location of the IPython directory has been determined, you need to know what filename to use for the configuration file. The basic idea is that each application has its own default configuration filename. The default named used by the **ipython** command line program is `ipython_config.py`. This value can be overridden by the `-config_file` command line flag. A sample `ipython_config.py` file can be found in `IPython.config.default.ipython_config.py`. Simple copy it to your IPython directory to begin using it.

6.1.6 Profiles

A profile is simply a configuration file that follows a simple naming convention and can be loaded using a simplified syntax. The idea is that users often want to maintain a set of configuration files for different purposes: one for doing numerical computing with NumPy and SciPy and another for doing symbolic computing with SymPy. Profiles make it easy to keep a separate configuration file for each of these purposes.

Let's start by showing how a profile is used:

```
$ ipython -p sympy
```

This tells the **ipython** command line program to get its configuration from the “sympy” profile. The search path for profiles is the same as that of regular configuration files. The only difference is that profiles are named in a special way. In the case above, the “sympy” profile would need to have the name `ipython_config_sympy.py`.

The general pattern is this: simply add `_profilename` to the end of the normal configuration file name. Then load the profile by adding `-p profilename` to your command line options.

IPython ships with some sample profiles in `IPython.config.profile`. Simply copy these to your IPython directory to begin using them.

6.1.7 Design requirements

Here are the main requirements we wanted our configuration system to have:

- Support for hierarchical configuration information.
- Full integration with command line option parsers. Often, you want to read a configuration file, but then override some of the values with command line options. Our configuration system automates this process and allows each command line option to be linked to a particular attribute in the configuration hierarchy that it will override.
- Configuration files that are themselves valid Python code. This accomplishes many things. First, it becomes possible to put logic in your configuration files that sets attributes based on your operating system, network setup, Python version, etc. Second, Python has a super simple syntax for accessing hierarchical data structures, namely regular attribute access (`Foo.Bar.Bam.name`). Third, using Python makes it easy for users to import configuration attributes from one configuration file to another. Forth, even though Python is dynamically typed, it does have types that can be checked at runtime. Thus, a `1` in a config file is the integer `'1'`, while a `'1'` is a string.
- A fully automated method for getting the configuration information to the classes that need it at runtime. Writing code that walks a configuration hierarchy to extract a particular attribute is painful. When you have complex configuration information with hundreds of attributes, this makes you want to cry.
- Type checking and validation that doesn't require the entire configuration hierarchy to be specified statically before runtime. Python is a very dynamic language and you don't always know everything that needs to be configured when a program starts.

6.2 IPython extensions

Configuration files are just the first level of customization that IPython supports. The next level is that of extensions. An IPython extension is an importable Python module that has a few special function. By defining these functions, users can customize IPython by accessing the actual runtime objects of IPython. Here is a sample extension:

```
# myextension.py

def load_ipython_extension(ipython):
    # The ``ipython`` argument is the currently active
    # :class:`InteractiveShell` instance that can be used in any way.
    # This allows you do to things like register new magics, plugins or
    # aliases.

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

This `load_ipython_extension()` function is called after your extension is imported and the currently active `InteractiveShell` instance is passed as the only argument. You can do anything you want with IPython at that point.

The `load_ipython_extension()` will be called again if you load or reload the extension again. It is up to the extension author to add code to manage that.

You can put your extension modules anywhere you want, as long as they can be imported by Python's standard import mechanism. However, to make it easy to write extensions, you can also put your extensions in `os.path.join(self.ipython_dir, 'extensions')`. This directory is added to `sys.path` automatically.

6.2.1 Using extensions

There are two ways you can tell IPython to use your extension:

1. Listing it in a configuration file.
2. Using the `%load_ext` magic function.

To load an extension called `myextension.py` add the following logic to your configuration file:

```
c.Global.extensions = [  
    'myextension'  
]
```

To load that same extension at runtime, use the `%load_ext` magic:

```
.. sourcecode:: ipython
```

```
In [1]: %load_ext myextension
```

To summarize, in conjunction with configuration files and profiles, IPython extensions give you complete and flexible control over your IPython setup.

6.3 IPython plugins

IPython has a plugin mechanism that allows users to create new and custom runtime components for IPython. Plugins are different from extensions:

- Extensions are used to load plugins.
- Extensions are a more advanced configuration system that gives you access to the running IPython instance.
- Plugins add entirely new capabilities to IPython.
- Plugins are traitled and configurable.

At this point, our plugin system is brand new and the documentation is minimal. If you are interested in creating a new plugin, see the following files:

- `IPython/extensions/parallelmagic.py`
- `IPython/extensions/pretty.`

As well as our documentation on the configuration system and extensions.

6.4 Configuring the ipython command line application

This section contains information about how to configure the **ipython** command line application. See the *configuration overview* for a more general description of the configuration system and configuration file format.

The default configuration file for the **ipython** command line application is `ipython_config.py`. By setting the attributes in this file, you can configure the application. A sample is provided in `IPython.config.default.ipython_config`. Simply copy this file to your IPython directory to start using it.

Most configuration attributes that this file accepts are associated with classes that are subclasses of `Component`.

A few configuration attributes are not associated with a particular `Component` subclass. These are application wide configuration attributes and are stored in the `Global` sub-configuration section. We begin with a description of these attributes.

6.4.1 Global configuration

Assuming that your configuration file has the following at the top:

```
c = get_config()
```

the following attributes can be set in the `Global` section.

- c.Global.display_banner** A boolean that determined if the banner is printer when **ipython** is started.
- c.Global.classic** A boolean that determines if IPython starts in “classic” mode. In this mode, the prompts and everything mimic that of the normal **python** shell
- c.Global.nosep** A boolean that determines if there should be no blank lines between prompts.
- c.Global.log_level** An integer that sets the detail of the logging level during the startup of **ipython**. The default is 30 and the possible values are (0, 10, 20, 30, 40, 50). Higher is quieter and lower is more verbose.
- c.Global.extensions** A list of strings, each of which is an importable IPython extension. An IPython extension is a regular Python module or package that has a `load_ipython_extension(ip)()` method. This method gets called when the extension is loaded with the currently running `InteractiveShell` as its only argument. You can put your extensions anywhere they can be imported but we add the `extensions` subdirectory of the `ipython` directory to `sys.path` during extension loading, so you can put them there as well. Extensions are not executed in the user’s interactive namespace and they must be pure Python code. Extensions are the recommended way of customizing **ipython**. Extensions can provide an `unload_ipython_extension()` that will be called when the extension is unloaded.
- c.Global.exec_lines** A list of strings, each of which is Python code that is run in the user’s namespace after IPython start. These lines can contain full IPython syntax with magics, etc.

c.Global.exec_files A list of strings, each of which is the full pathname of a `.py` or `.ipy` file that will be executed as IPython starts. These files are run in IPython in the user's namespace. Files with a `.py` extension need to be pure Python. Files with a `.ipy` extension can have custom IPython syntax (magics, etc.). These files need to be in the `cwd`, the `ipythondir` or be absolute paths.

6.4.2 Classes that can be configured

The following classes can also be configured in the configuration file for **ipython**:

- `InteractiveShell`
- `PrefilterManager`
- `AliasManager`

To see which attributes of these classes are configurable, please see the source code for these classes, the class docstrings or the sample configuration file `IPython.config.default.ipython_config`.

6.4.3 Example

For those who want to get a quick start, here is a sample `ipython_config.py` that sets some of the common configuration attributes:

```
# sample ipython_config.py
c = get_config()

c.Global.display_banner = True
c.Global.log_level = 20
c.Global.extensions = [
    'myextension'
]
c.Global.exec_lines = [
    'import numpy',
    'import scipy'
]
c.Global.exec_files = [
    'mycode.py',
    'fancy.ipy'
]
c.InteractiveShell.autoindent = True
c.InteractiveShell.colors = 'LightBG'
c.InteractiveShell.confirm_exit = False
c.InteractiveShell.deep_reload = True
c.InteractiveShell.editor = 'nano'
c.InteractiveShell.prompt_in1 = 'In [#]: '
c.InteractiveShell.prompt_in2 = '    .\D.: '
c.InteractiveShell.prompt_out = 'Out[#]: '
c.InteractiveShell.prompts_pad_left = True
c.InteractiveShell.xmode = 'Context'

c.PrefilterManager.multi_line_specials = True
```

```
c.AliasManager.user_aliases = [  
    ('la', 'ls -al')  
]
```

6.5 Editor configuration

IPython can integrate with text editors in a number of different ways:

- Editors (such as (X)Emacs [Emacs], vim [vim] and TextMate [TextMate]) can send code to IPython for execution.
- IPython's `%edit` magic command can open an editor of choice to edit a code block.

The `%edit` command (and its alias `%ed`) will invoke the editor set in your environment as `EDITOR`. If this variable is not set, it will default to `vi` under Linux/Unix and to `notepad` under Windows. You may want to set this variable properly and to a lightweight editor which doesn't take too long to start (that is, something other than a new instance of Emacs). This way you can edit multi-line code quickly and with the power of a real editor right inside IPython.

You can also control the editor via the command-line option `'-editor'` or in your configuration file, by setting the `InteractiveShell.editor` configuration attribute.

6.5.1 TextMate

Currently, TextMate support in IPython is broken. It used to work well, but the code has been moved to `IPython.quarantine` until it is updated.

6.5.2 vim configuration

Currently, vim support in IPython is broken. Like the TextMate code, the vim support code has been moved to `IPython.quarantine` until it is updated.

6.5.3 (X)Emacs

6.5.4 Editor

If you are a dedicated Emacs user, and want to use Emacs when IPython's `%edit` magic command is called you should set up the Emacs server so that new requests are handled by the original process. This means that almost no time is spent in handling the request (assuming an Emacs process is already running). For this to work, you need to set your `EDITOR` environment variable to `'emacsclient'`. The code below, supplied by Francois Pinard, can then be used in your `.emacs` file to enable the server:

```
(defvar server-buffer-clients)  
(when (and (fboundp 'server-start) (string-equal (getenv "TERM") 'xterm))  
  (server-start)  
  (defun fp-kill-server-with-buffer-routine ()
```

```
(and server-buffer-clients (server-done)))  
(add-hook 'kill-buffer-hook 'fp-kill-server-with-buffer-routine))
```

Thanks to the work of Alexander Schmolck and Prabhu Ramachandran, currently (X)Emacs and IPython get along very well in other ways.

Note: You will need to use a recent enough version of `python-mode.el`, along with the file `ipython.el`. You can check that the version you have of `python-mode.el` is new enough by either looking at the revision number in the file itself, or asking for it in (X)Emacs via `M-x py-version`. Versions 4.68 and newer contain the necessary fixes for proper IPython support.

The file `ipython.el` is included with the IPython distribution, in the directory `docs/emacs`. Once you put these files in your Emacs path, all you need in your `.emacs` file is:

```
(require 'ipython)
```

This should give you full support for executing code snippets via IPython, opening IPython as your Python shell via `C-c !`, etc.

You can customize the arguments passed to the IPython instance at startup by setting the `py-python-command-args` variable. For example, to start always in `pylab` mode with hardcoded light-background colors, you can use:

```
(setq py-python-command-args '("-pylab" "-colors" "LightBG"))
```

If you happen to get garbage instead of colored prompts as described in the previous section, you may need to set also in your `.emacs` file:

```
(setq ansi-color-for-comint-mode t)
```

Notes on emacs support:

- There is one caveat you should be aware of: you must start the IPython shell before attempting to execute any code regions via `C-c |`. Simply type `C-c !` to start IPython before passing any code regions to the interpreter, and you shouldn't experience any problems. This is due to a bug in Python itself, which has been fixed for Python 2.3, but exists as of Python 2.2.2 (reported as SF bug [737947]).
- The (X)Emacs support is maintained by Alexander Schmolck, so all comments/requests should be directed to him through the IPython mailing lists.
- This code is still somewhat experimental so it's a bit rough around the edges (although in practice, it works quite well).
- Be aware that if you customized `py-python-command` previously, this value will override what `ipython.el` does (because loading the customization variables comes later).

6.6 Outdated configuration information that might still be useful

Warning: All of the information in this file is outdated. Until the new configuration system is better documented, this material is being kept.

This section will help you set various things in your environment for your IPython sessions to be as efficient as possible. All of IPython's configuration information, along with several example files, is stored in a directory named by default `$HOME/.ipython`. You can change this by defining the environment variable `IPYTHONDIR`, or at runtime with the command line option `-ipythondir`.

If all goes well, the first time you run IPython it should automatically create a user copy of the config directory for you, based on its builtin defaults. You can look at the files it creates to learn more about configuring the system. The main file you will modify to configure IPython's behavior is called `ipythonrc` (with a `.ini` extension under Windows), included for reference *here*. This file is very commented and has many variables you can change to suit your taste, you can find more details *here*. Here we discuss the basic things you will want to make sure things are working properly from the beginning.

6.6.1 Color

The default IPython configuration has most bells and whistles turned on (they're pretty safe). But there's one that may cause problems on some systems: the use of color on screen for displaying information. This is very useful, since IPython can show prompts and exception tracebacks with various colors, display syntax-highlighted source code, and in general make it easier to visually parse information.

The following terminals seem to handle the color sequences fine:

- Linux main text console, KDE Konsole, Gnome Terminal, E-term, `rxvt`, `xterm`.
- CDE terminal (tested under Solaris). This one boldfaces light colors.
- (X)Emacs buffers. See the **emacs** section for more details on using IPython with (X)Emacs.
- A Windows (XP/2k) command prompt with `pyreadline`.
- A Windows (XP/2k) CygWin shell. Although some users have reported problems; it is not clear whether there is an issue for everyone or only under specific configurations. If you have full color support under cygwin, please post to the IPython mailing list so this issue can be resolved for all users.

These have shown problems:

- Windows command prompt in WinXP/2k logged into a Linux machine via telnet or ssh.
- Windows native command prompt in WinXP/2k, without Gary Bishop's extensions. Once Gary's readline library is installed, the normal WinXP/2k command prompt works perfectly.

Currently the following color schemes are available:

- NoColor: uses no color escapes at all (all escapes are empty `""` strings). This 'scheme' is thus fully safe to use in any terminal.
- Linux: works well in Linux console type environments: dark background with light fonts. It uses bright colors for information, so it is difficult to read if you have a light colored background.

- **LightBG**: the basic colors are similar to those in the Linux scheme but darker. It is easy to read in terminals with light backgrounds.

IPython uses colors for two main groups of things: prompts and tracebacks which are directly printed to the terminal, and the object introspection system which passes large sets of data through a pager.

6.6.2 Input/Output prompts and exception tracebacks

You can test whether the colored prompts and tracebacks work on your system interactively by typing ‘`%colors Linux`’ at the prompt (use ‘`%colors LightBG`’ if your terminal has a light background). If the input prompt shows garbage like:

```
[0;32mIn [[1;32m1[0;32m]: [0;00m
```

instead of (in color) something like:

```
In [1]:
```

this means that your terminal doesn’t properly handle color escape sequences. You can go to a ‘no color’ mode by typing ‘`%colors NoColor`’.

You can try using a different terminal emulator program (Emacs users, see below). To permanently set your color preferences, edit the file `$HOME/.ipython/ipythonrc` and set the `colors` option to the desired value.

6.6.3 Object details (types, docstrings, source code, etc.)

IPython has a set of special functions for studying the objects you are working with, discussed in detail [here](#). But this system relies on passing information which is longer than your screen through a data pager, such as the common Unix `less` and `more` programs. In order to be able to see this information in color, your pager needs to be properly configured. I strongly recommend using `less` instead of `more`, as it seems that `more` simply can not understand colored text correctly.

In order to configure `less` as your default pager, do the following:

1. Set the environment `PAGER` variable to `less`.
2. Set the environment `LESS` variable to `-r` (plus any other options you always want to pass to `less` by default). This tells `less` to properly interpret control sequences, which is how color information is given to your terminal.

For the bash shell, add to your `~/.bashrc` file the lines:

```
export PAGER=less
export LESS=-r
```

For the csh or tcsh shells, add to your `~/.cshrc` file the lines:

```
setenv PAGER less
setenv LESS -r
```

There is similar syntax for other Unix shells, look at your system documentation for details.

If you are on a system which lacks proper data pagers (such as Windows), IPython will use a very limited builtin pager.

6.6.4 Fine-tuning your prompt

IPython's prompts can be customized using a syntax similar to that of the bash shell. Many of bash's escapes are supported, as well as a few additional ones. We list them below:

```
\#
    the prompt/history count number. This escape is automatically
    wrapped in the coloring codes for the currently active color scheme.
\N
    the 'naked' prompt/history count number: this is just the number
    itself, without any coloring applied to it. This lets you produce
    numbered prompts with your own colors.
\D
    the prompt/history count, with the actual digits replaced by dots.
    Used mainly in continuation prompts (prompt_in2)
\w
    the current working directory
\W
    the basename of current working directory
\Xn
    where $n=0\ldots5.$ The current working directory, with $HOME
    replaced by ~, and filtered out to contain only $n$ path elements
\Yn
    Similar to \Xn, but with the $n+1$ element included if it is ~ (this
    is similar to the behavior of the %cn escapes in tcsh)
\u
    the username of the current user
\$$
    if the effective UID is 0, a #, otherwise a $
\h
    the hostname up to the first '.'
\H
    the hostname
\n
    a newline
\r
    a carriage return
\v
    IPython version string
```

In addition to these, ANSI color escapes can be inserted into the prompts, as `C_ColorName`. The list of valid color names is: Black, Blue, Brown, Cyan, DarkGray, Green, LightBlue, LightCyan, LightGray, LightGreen, LightPurple, LightRed, NoColor, Normal, Purple, Red, White, Yellow.

Finally, IPython supports the evaluation of arbitrary expressions in your prompt string. The prompt strings are evaluated through the syntax of PEP 215, but basically you can use `$x.y` to expand the value of `x.y`, and for more complicated expressions you can use braces: `${foo()}+x` will call function `foo` and add to it the value of `x`, before putting the result into your prompt. For example, using `prompt_in1` `'${com-mands.getoutput("uptime")}\nIn [#]: '` will print the result of the `uptime` command on each prompt (assuming

the commands module has been imported in your ipythonrc file).

Prompt examples

The following options in an ipythonrc file will give you IPython's default prompts:

```
prompt_in1 'In [\#]:'  
prompt_in2 '    .\D.:'  
prompt_out 'Out [\#]:'
```

which look like this:

```
In [1]: 1+2  
Out[1]: 3  
  
In [2]: for i in (1,2,3):  
...:     print i,  
...:  
1 2 3
```

These will give you a very colorful prompt with path information:

```
#prompt_in1 '\C_Red\u\C_Blue[\C_Cyan\Y1\C_Blue]\C_LightGreen\#>'  
prompt_in2 ' ..\D>'  
prompt_out '<\#>'
```

which look like this:

```
fperez[~/ipython]1> 1+2  
                  <1> 3  
fperez[~/ipython]2> for i in (1,2,3):  
...>          print i,  
...>  
1 2 3
```


IPYTHON DEVELOPER'S GUIDE

7.1 How to contribute to IPython

7.1.1 Overview

IPython development is done using Git [\[Git\]](#) and Github.com [\[Github.com\]](#). This makes it easy for people to contribute to the development of IPython. There are several ways in which you can join in.

7.1.2 Merging a branch into trunk

Core developers, who ultimately merge any approved branch (from themselves, another developer, or any third-party contribution) will typically use **git merge** to merge the branch into the trunk and push it to the main Git repository. There are a number of things to keep in mind when doing this, so that the project history is easy to understand in the long run, and that generating release notes is as painless and accurate as possible.

- When you merge any non-trivial functionality (from one small bug fix to a big feature branch), please remember to always edit the appropriate file in the *What's new* section of our documentation. Ideally, the author of the branch should provide this content when they submit the branch for review. But if they don't it is the responsibility of the developer doing the merge to add this information.
- When merges are done, the practice of putting a summary commit message in the merge is *extremely* useful. It is probably easiest if you simply use the same list of changes that were added to the *What's new* section of the documentation.
- It's important that we remember to always credit who gave us something if it's not the committer. In general, we have been fairly good on this front, this is just a reminder to keep things up. As a note, if you are ever committing something that is completely (or almost so) a third-party contribution, do the commit as:

```
$ git commit --author="Someone Else"
```

This way it will show that name separately in the log, which makes it even easier to spot. Obviously we often rework third party contributions extensively, but this is still good to keep in mind for cases when we don't touch the code too much.

7.2 Working with *ipython* source code

Contents:

7.2.1 Introduction

These pages describe a [git](#) and [github](#) workflow for the *ipython* project.

There are several different workflows here, for different ways of working with *ipython*.

This is not a comprehensive [git](#) reference, it's just a workflow for our own project. It's tailored to the [github](#) hosting service. You may well find better or quicker ways of getting stuff done with [git](#), but these should get you started.

For general resources for learning [git](#) see [git resources](#).

7.2.2 Install git

Overview

Debian / Ubuntu	<code>sudo apt-get install git-core</code>
Fedora	<code>sudo yum install git-core</code>
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the [git](#) page for the most recent information.

Have a look at the [github](#) install help pages available from [github help](#)

There are good instructions here: http://book.git-scm.com/2_installing_git.html

7.2.3 Following the latest source

These are the instructions if you just want to follow the latest *ipython* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the git repository from [github](#)
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/ipython/ipython.git
```

You now have a copy of the code tree in the new `ipython` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd ipython
git pull
```

The tree in `ipython` will now have the latest changes from the initial repository.

7.2.4 Making a patch

You've discovered a bug or something else you want to change in `ipython` - excellent!

You've worked out a way to fix it - even better!

You want to tell us about it - best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/ipython/ipython.git
# make a branch for your patching
cd ipython
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

```
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the [ipython mailing list](#) - where we will thank you warmly.

In detail

1. Tell `git` who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the [ipython](#) repository:

```
git clone git://github.com/ipython/ipython.git
cd ipython
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag - you can just take on faith - or see [why the -a flag?](#).

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the master branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the [ipython mailing list](#).

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the [ipython](#) repository on [github](#) - *Making your own copy (fork) of ipython*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/ipython.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

7.2.5 Git for development

Contents:

Making your own copy (fork) of ipython

You need to do this only once. The instructions here are very similar to the instructions at <http://help.github.com/forking/> - please see that page for more detail. We're repeating some of it here just to give the specifics for the [ipython](#) project, and to suggest some default names.

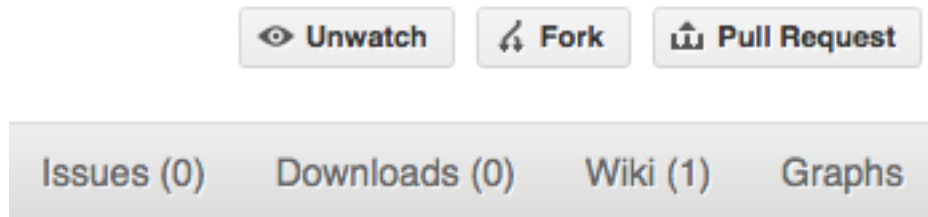
Set up and configure a github account

If you don't have a [github](#) account, go to the [github](#) page, and make one.

You then need to configure your account to allow write access - see the [Generating SSH keys](#) help on [github help](#).

Create your own forked copy of ipython

1. Log into your [github](#) account.
2. Go to the [ipython](#) github home at [ipython github](#).
3. Click on the *fork* button:



Now, after a short pause and some ‘Hardcore forking action’, you should find yourself at the home page for your own forked copy of [ipython](#).

Set up your fork

First you follow the instructions for *Making your own copy (fork) of ipython*.

Overview

```
git clone git@github.com:your-user-name/ipython.git
cd ipython
git remote add upstream git://github.com/ipython/ipython.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/ipython.git`
2. Investigate. Change directory to your new repo: `cd ipython`. Then `git branch -a` to show you all branches. You’ll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the master branch, and that you also have a remote connection to origin/master. What remote repository is remote/origin? Try `git remote -v` to see the URLs for the remote. They will point to your [github](#) fork.

Now you want to connect to the upstream [ipython github](#) repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd ipython
git remote add upstream git://github.com/ipython/ipython.git
```

`upstream` here is just the arbitrary name we’re using to refer to the main [ipython](#) repository at [ipython github](#).

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream      git://github.com/ipython/ipython.git (fetch)
upstream      git://github.com/ipython/ipython.git (push)
origin        git@github.com:your-user-name/ipython.git (fetch)
origin        git@github.com:your-user-name/ipython.git (push)
```

Configure git

Overview

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

In detail

This is to tell `git` who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

This will write the settings into your git configuration file - a file called `.gitconfig` in your home directory.

Advanced git configuration

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`.

The easiest way to do this, is to create a `.gitconfig` file in your home directory, with contents like this:

```
[core]
    editor = emacs
[user]
    email = you@yourdomain.example.com
    name = Your Name Comes Here
[alias]
    st = status
    stat = status
    co = checkout
[color]
```

```
diff = auto
status = true
```

(of course you'll need to set your email and name, and may want to set your editor). If you prefer, you can do the same thing from the command line:

```
git config --global core.editor emacs
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
git config --global alias.st status
git config --global alias.stat status
git config --global alias.co checkout
git config --global color.diff auto
git config --global color.status true
```

These commands will write to your user's git configuration file `~/.gitconfig`.

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

Other configuration recommended by Yarik

In your `~/.gitconfig` file alias section:

```
wdiff = diff --color-words
```

so that `git wdiff` gives a nicely formatted output of the diff.

To enforce summaries when doing merges(`~/.gitconfig` file again):

```
[merge]
    summary = true
```

Development workflow

You already have your own forked copy of the `ipython` repository, by following *Making your own copy (fork) of ipython*, *Set up your fork*, and you have configured `git` by following *Configure git*.

Workflow summary

- Keep your `master` branch clean of edits that have not been merged to the main `ipython` development repo. Your `master` then will follow the main `ipython` repository.
- Start a new *feature branch* for each set of edits that you do.
- If you can avoid it, try not to merge other branches into your feature branch while you are working.
- Ask for review!

This way of working really helps to keep work well organized, and in keeping history as clear as possible.

See - for example - [linux git workflow](#).

Making a new feature branch

```
git branch my-new-feature
git checkout my-new-feature
```

Generally, you will want to keep this also on your public [github](#) fork of [ipython](#). To do this, you `git push` this new branch up to your [github](#) repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your [github](#) repo, called `origin`. You push up to your own repo on [github](#) with:

```
git push origin my-new-feature
```

From now on [git](#) will know that `my-new-feature` is related to the `my-new-feature` branch in the [github](#) repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see [git status](#)). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` ([git diff](#)).
4. Add any new files to version control `git add new_file_name` (see [git add](#)).
5. To commit all modified files into the local copy of your repo., do `git commit -am 'A commit message'`. Note the `-am` options to commit. The `m` flag just signals that you're going to type a message on the command line. The `a` flag - you can just take on faith - or see [why the -a flag?](#). See also the [git commit](#) manual page.

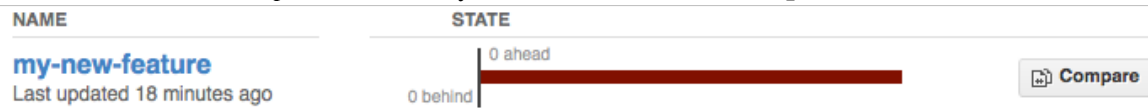
6. To push the changes up to your forked repo on [github](#), do a `git push` (see *git push*).

Asking for code review

1. Go to your repo URL - e.g. `http://github.com/your-user-name/ipython`.
2. Click on the *Branch list* button:



3. Click on the *Compare* button for your feature branch - here `my-new-feature`:



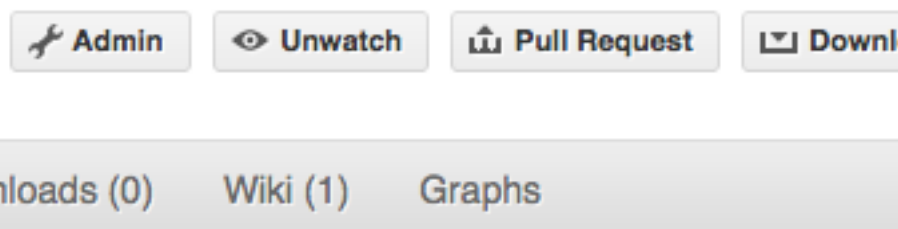
4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the [ipython mailing list](#), asking for review. The URL will look something like: `http://github.com/your-user-name/ipython/compare/master...my-new-feature`. There's an example at [http://github.com/matthew-brett/nipy/compare/master...find-install-data](#) See: [http://github.com/blog/612-introducing-github-compare-view](#) for more detail.

The generated comparison, is between your feature branch `my-new-feature`, and the place in `master` from which you branched `my-new-feature`. In other words, you can keep updating `master` without interfering with the output from the comparison. More detail? Note the three dots in the URL above (`master...my-new-feature`) and see *dot2-dot3*.

Asking for your changes to be merged with the main repo

When you are ready to ask for the merge of your code:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/ipython.git`.
2. Click on the 'Pull request' button:



Enter a message; we suggest you select only `ipython` as the recipient. The message will go to the [ipython mailing list](#). Please feel free to add others from the list as you like.

Merging from trunk

This updates your code from the upstream [ipython github](#) repo.

Overview

```
# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge upstream/master
```

In detail We suggest that you do this only for your `master` branch, and leave your ‘feature’ branches unmerged, to keep their history as clean as possible. This makes code review easier:

```
git checkout master
```

Make sure you have done [Linking your repository to the upstream repo](#).

Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

```
git fetch upstream
```

then merging into your current branch:

```
git merge upstream/master
```

Deleting a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

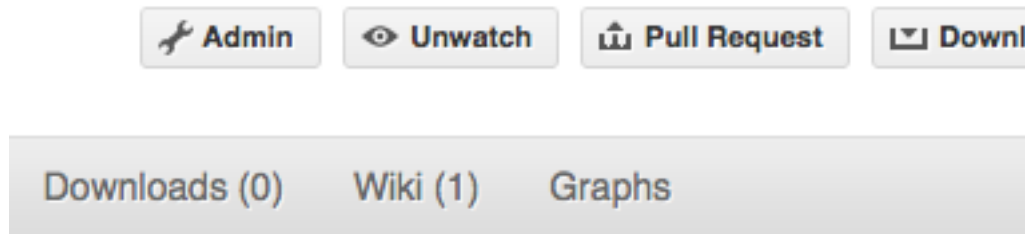
(Note the colon `:` before `test-branch`. See also: <http://github.com/guides/remove-a-remote-branch>)

Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via [github](#).

First fork `ipython` into your account, as from [Making your own copy \(fork\) of ipython](#).

Then, go to your forked repository github page, say `http://github.com/your-user-name/ipython`. Click on the ‘Admin’ button, and add anyone else to the repo as a collaborator:



Now all those people can do:

```
git clone git@github.com:your-user-name/ipython.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Exploring your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the [network graph visualizer](#) for your [github](#) repo.

7.2.6 git resources

Tutorials and summaries

- [github help](#) has an excellent series of how-to guides.
- [learn.github](#) has an excellent series of tutorials
- The [pro git book](#) is a good in-depth book on git.
- A [git cheat sheet](#) is a page giving summaries of common commands.
- The [git user manual](#)
- The [git tutorial](#)
- The [git community book](#)

- [git ready](#) - a nice series of tutorials
- [git casts](#) - video snippets giving git how-tos.
- [git magic](#) - extended introduction with intermediate detail
- Fernando Perez' git page - [Fernando's git page](#) - many links and tips
- A good but technical page on [git concepts](#)
- Th [git parable](#) is an easy read explaining the concepts behind git.
- [git svn crash course](#): [git](#) for those of us used to [subversion](#)

Advanced git workflow

There are many ways of working with [git](#); here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on [git management](#)
- Linus Torvalds on [linux git workflow](#) . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- [git add](#)
- [git branch](#)
- [git checkout](#)
- [git clone](#)
- [git commit](#)
- [git config](#)
- [git diff](#)
- [git log](#)
- [git pull](#)
- [git push](#)
- [git remote](#)
- [git status](#)

7.3 Coding guide

7.3.1 General coding conventions

In general, we'll try to follow the standard Python style conventions as described in Python's PEP 8 [PEP8], the official Python Style Guide.

Other general comments:

- In a large file, top level classes and functions should be separated by 2 lines to make it easier to separate them visually.
- Use 4 spaces for indentation, **never** use hard tabs.
- Keep the ordering of methods the same in classes that have the same methods. This is particularly true for classes that implement similar interfaces and for interfaces that are similar.

7.3.2 Naming conventions

In terms of naming conventions, we'll follow the guidelines of PEP 8 [PEP8]. Some of the existing code doesn't honor this perfectly, but for all new IPython code (and much existing code is being refactored), we'll use:

- All lowercase module names.
- CamelCase for class names.
- lowercase_with_underscores for methods, functions, variables and attributes.

This may be confusing as some of the existing codebase uses a different convention (lowerCamelCase for methods and attributes). Slowly, we will move IPython over to the new convention, providing shadow names for backward compatibility in public interfaces.

There are, however, some important exceptions to these rules. In some cases, IPython code will interface with packages (Twisted, Wx, Qt) that use other conventions. At some level this makes it impossible to adhere to our own standards at all times. In particular, when subclassing classes that use other naming conventions, you must follow their naming conventions. To deal with cases like this, we propose the following policy:

- If you are subclassing a class that uses different conventions, use its naming conventions throughout your subclass. Thus, if you are creating a Twisted Protocol class, used Twisted's `namingSchemeForMethodsAndAttributes`.
- All IPython's official interfaces should use our conventions. In some cases this will mean that you need to provide shadow names (first implement `fooBar` and then `foo_bar = fooBar`). We want to avoid this at all costs, but it will probably be necessary at times. But, please use this sparingly!

Implementation-specific *private* methods will use `_single_underscore_prefix`. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).

Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like `runlines()` where `runsource()` and `runcode()` had established precedent).

The old IPython codebase has a big mix of classes and modules prefixed with an explicit `IP`. In Python this is mostly unnecessary, redundant and frowned upon, as namespaces offer cleaner prefixing. The only case where this approach is justified is for classes which are expected to be imported into external namespaces and a very generic name (like `Shell`) is too likely to clash with something else. However, if a prefix seems absolutely necessary the more specific `IPY` or `ipy` are preferred.

7.3.3 Attribute declarations for objects

In general, objects should declare in their *class* all attributes the object is meant to hold throughout its life. While Python allows you to add an attribute to an instance at any point in time, this makes the code harder to read and requires methods to constantly use checks with `hasattr()` or `try/except` calls. By declaring all attributes of the object in the class header, there is a single place one can refer to for understanding the object's data interface, where comments can explain the role of each variable and when possible, sensible defaults can be assigned.

Warning: If an attribute is meant to contain a mutable object, it should be set to `None` in the class and its mutable value should be set in the object's constructor. Since class attributes are shared by all instances, failure to do this can lead to difficult to track bugs. But you should still set it in the class declaration so the interface specification is complete and documented in one place.

A simple example:

```
class foo:
    # X does..., sensible default given:
    x = 1
    # y does..., default will be set by constructor
    y = None
    # z starts as an empty list, must be set in constructor
    z = None

    def __init__(self, y):
        self.y = y
        self.z = []
```

7.3.4 New files

When starting a new file for IPython, you can use the following template as a starting point that has a few common things pre-written for you. The template is included in the documentation sources as `docs/sources/development/template.py`:

```
"""A one-line description.

A longer description that spans multiple lines. Explain the purpose of the
file and provide a short list of the key classes/functions it contains. This
is the docstring shown when some does 'import foo;foo?' in IPython, so it
should be reasonably useful and informative.
"""
#-----
# Copyright (c) 2010, IPython Development Team.
```

```
#
# Distributed under the terms of the Modified BSD License.
#
# The full license is in the file COPYING.txt, distributed with this software.
#-----

#-----
# Imports
#-----

from __future__ import print_function

# [remove this comment in production]
#
# List all imports, sorted within each section (stdlib/third-party/ipython).
# For 'import foo', use one import per line. For 'from foo.bar import a, b, c'
# it's OK to import multiple items, use the parenthesized syntax 'from foo
# import (a, b, ...)' if the list needs multiple lines.

# Stdlib imports

# Third-party imports

# Our own imports

# [remove this comment in production]
#
# Use broad section headers like this one that make it easier to navigate the
# file, with descriptive titles. For complex classes, simliar (but indented)
# headers are useful to organize the internal class structure.

#-----
# Globals and constants
#-----

#-----
# Local utilities
#-----

#-----
# Classes and functions
#-----
```

7.4 Documenting IPython

When contributing code to IPython, you should strive for clarity and consistency, without falling prey to a style straitjacket. Basically, ‘document everything, try to be consistent, do what makes sense.’

By and large we follow existing Python practices in major projects like Python itself or NumPy, this document provides some additional detail for IPython.

7.4.1 Standalone documentation

All standalone documentation should be written in plain text (`.txt`) files using reStructuredText [reStructuredText] for markup and formatting. All such documentation should be placed in the directory `docs/source` of the IPython source tree. Or, when appropriate, a suitably named subdirectory should be used. The documentation in this location will serve as the main source for IPython documentation.

The actual HTML and PDF docs are built using the Sphinx [Sphinx] documentation generation tool. Once you have Sphinx installed, you can build the html docs yourself by doing:

```
$ cd ipython-mybranch/docs
$ make html
```

Our usage of Sphinx follows that of matplotlib [Matplotlib] closely. We are using a number of Sphinx tools and extensions written by the matplotlib team and will mostly follow their conventions, which are nicely spelled out in their documentation guide [MatplotlibDocGuide]. What follows is thus a abridged version of the matplotlib documentation guide, taken with permission from the matplotlib team.

If you are reading this in a web browser, you can click on the “Show Source” link to see the original reStructuredText for the following examples.

A bit of Python code:

```
for i in range(10):
    print i,
print "A big number:", 2**34
```

An interactive Python session:

```
>>> from IPython.utils.path import get_ipython_dir
>>> get_ipython_dir()
'/home/fperez/.ipython'
```

An IPython session:

```
In [7]: import IPython

In [8]: print "This IPython is version:", IPython.__version__
This IPython is version: 0.9.1

In [9]: 2+4
Out[9]: 6
```

A bit of shell code:

```
cd /tmp
echo "My home directory is: $HOME"
ls
```

7.4.2 Docstring format

Good docstrings are very important. Unfortunately, Python itself only provides a rather loose standard for docstrings [PEP257], and there is no universally accepted convention for all the different parts of a complete

docstring. However, the NumPy project has established a very reasonable standard, and has developed some tools to support the smooth inclusion of such docstrings in Sphinx-generated manuals. Rather than inventing yet another pseudo-standard, IPython will be henceforth documented using the NumPy conventions; we carry copies of some of the NumPy support tools to remain self-contained, but share back upstream with NumPy any improvements or fixes we may make to the tools.

The NumPy documentation guidelines [\[NumPyDocGuide\]](#) contain detailed information on this standard, and for a quick overview, the NumPy example docstring [\[NumPyExampleDocstring\]](#) is a useful read.

For user-facing APIs, we try to be fairly strict about following the above standards (even though they mean more verbose and detailed docstrings). Wherever you can reasonably expect people to do introspection with:

```
In [1]: some_function?
```

the docstring should follow the NumPy style and be fairly detailed.

For purely internal methods that are only likely to be read by others extending IPython itself we are a bit more relaxed, especially for small/short methods and functions whose intent is reasonably obvious. We still expect docstrings to be written, but they can be simpler. For very short functions with a single-line docstring you can use something like:

```
def add(a, b):  
    """The sum of two numbers.  
    """  
    code
```

and for longer multiline strings:

```
def add(a, b):  
    """The sum of two numbers.  
  
    Here is the rest of the docs.  
    """  
    code
```

Here are two additional PEPs of interest regarding documentation of code. While both of these were rejected, the ideas therein form much of the basis of docutils (the machinery to process reStructuredText):

- [Docstring Processing System Framework](#)
- [Docutils Design Specification](#)

Note: In the past IPython used epydoc so currently many docstrings still use epydoc conventions. We will update them as we go, but all new code should be documented using the NumPy standard.

7.5 Testing IPython for users and developers

7.5.1 Overview

It is extremely important that all code contributed to IPython has tests. Tests should be written as unittests, doctests or other entities that the IPython test system can detect. See below for more details on this.

Each subpackage in IPython should have its own `tests` directory that contains all of the tests for that subpackage. All of the files in the `tests` directory should have the word “tests” in them to enable the testing framework to find them.

In docstrings, examples (either using IPython prompts like `In [1]:` or ‘classic’ python `>>>` ones) can and should be included. The testing system will detect them as doctests and will run them; it offers control to skip parts or all of a specific doctest if the example is meant to be informative but shows non-reproducible information (like filesystem data).

If a subpackage has any dependencies beyond the Python standard library, the tests for that subpackage should be skipped if the dependencies are not found. This is very important so users don’t get tests failing simply because they don’t have dependencies.

The testing system we use is a hybrid of `nose` and Twisted’s `trial` test runner. We use both because `nose` detects more things than Twisted and allows for more flexible (and lighter-weight) ways of writing tests; in particular we’ve developed a `nose` plugin that allows us to paste verbatim IPython sessions and test them as doctests, which is extremely important for us. But the parts of IPython that depend on Twisted must be tested using `trial`, because only `trial` manages the Twisted reactor correctly.

7.5.2 For the impatient: running the tests

You can run IPython from the source download directory without even installing it system-wide or having configure anything, by typing at the terminal:

```
python ipython.py
```

In order to run the test suite, you must at least be able to import IPython, even if you haven’t fully installed the user-facing scripts yet (common in a development environment). You can then run the tests with:

```
python -c "import IPython; IPython.test()"
```

Once you have installed IPython either via a full install or using:

```
python setup.py develop
```

you will have available a system-wide script called `iptest` that runs the full test suite. You can then run the suite with:

```
iptest [args]
```

Regardless of how you run things, you should eventually see something like:

```
*****
Test suite completed for system with the following information:
{'commit_hash': '144fdae',
 'commit_source': 'repository',
 'ipython_path': '/home/fperez/usr/lib/python2.6/site-packages/IPython',
 'ipython_version': '0.11.dev',
 'os_name': 'posix',
 'platform': 'Linux-2.6.35-22-generic-i686-with-Ubuntu-10.10-maverick',
 'sys_executable': '/usr/bin/python',
 'sys_platform': 'linux2',
 'sys_version': '2.6.6 (r266:84292, Sep 15 2010, 15:52:39) \n[GCC 4.4.5]'}
```

Tools and libraries available at `test time`:

```
curses foolscap gobject gtk pexpect twisted wx wx.aui zope.interface
```

```
Ran 9 test groups in 67.213s
```

Status:

OK

If not, there will be a message indicating which test group failed and how to rerun that group individually. For example, this tests the `IPython.utils` subpackage, the `-v` option shows progress indicators:

```
$ iptest -v IPython.utils
.....SS..SSS.....S.S...
.....
-----
```

```
Ran 125 tests in 0.119s
```

OK (`SKIP=7`)

Because the IPython test machinery is based on nose, you can use all nose options and syntax, typing `iptest -h` shows all available options. For example, this lets you run the specific test `test_rehashx()` inside the `test_magic` module:

```
$ iptest -vv IPython.core.tests.test_magic:test_rehashx
IPython.core.tests.test_magic.test_rehashx(True,) ... ok
IPython.core.tests.test_magic.test_rehashx(True,) ... ok
-----
```

```
Ran 2 tests in 0.100s
```

OK

When developing, the `--pdb` and `--pdb-failures` of nose are particularly useful, these drop you into an interactive `pdb` session at the point of the error or failure respectively.

To run Twisted-using tests, use the **trial** command on a per file or package basis:

```
trial IPython.kernel
```

Note: The system information summary printed above is accessible from the top level package. If you encounter a problem with IPython, it's useful to include this information when reporting on the mailing list; use:

```
from IPython import sys_info
print sys_info()
```

and include the resulting information in your query.

7.5.3 For developers: writing tests

By now IPython has a reasonable test suite, so the best way to see what's available is to look at the `tests` directory in most subpackages. But here are a few pointers to make the process easier.

Main tools: `IPython.testing`

The `IPython.testing` package is where all of the machinery to test IPython (rather than the tests for its various parts) lives. In particular, the `iptest` module in there has all the smarts to control the test process. In there, the `make_exclude()` function is used to build a blacklist of exclusions, these are modules that do not get even imported for tests. This is important so that things that would fail to even import because of missing dependencies don't give errors to end users, as we stated above.

The `decorators` module contains a lot of useful decorators, especially useful to mark individual tests that should be skipped under certain conditions (rather than blacklisting the package altogether because of a missing major dependency).

Our nose plugin for doctests

The `plugin` subpackage in `testing` contains a nose plugin called `ipdoctest` that teaches nose about IPython syntax, so you can write doctests with IPython prompts. You can also mark doctest output with `# random` for the output corresponding to a single input to be ignored (stronger than using ellipsis and useful to keep it as an example). If you want the entire docstring to be executed but none of the output from any input to be checked, you can use the `# all-random` marker. The `IPython.testing.plugin.dtxample` module contains examples of how to use these; for reference here is how to use `# random`:

```
def ranfunc():
    """A function with some random output.

    Normal examples are verified as usual:
    >>> 1+3
    4

    But if you put '# random' in the output, it is ignored:
    >>> 1+3
    junk goes here... # random

    >>> 1+2
    again, anything goes #random
    if multiline, the random mark is only needed once.

    >>> 1+2
    You can also put the random marker at the end:
    # random

    >>> 1+2
    # random
    .. or at the beginning.
```

```
    More correct input is properly verified:
    >>> ranfunc()
    'ranfunc'
    """
    return 'ranfunc'
```

and an example of # all-random:

```
def random_all():
    """A function where we ignore the output of ALL examples.

    Examples:

    # all-random

    This mark tells the testing machinery that all subsequent examples
    should be treated as random (ignoring their output). They are still
    executed, so if a they raise an error, it will be detected as such,
    but their output is completely ignored.

    >>> 1+3
    junk goes here...

    >>> 1+3
    klasdfj;

    In [8]: print 'hello'
    world # random

    In [9]: iprand()
    Out[9]: 'iprand'
    """
    return 'iprand'
```

When writing docstrings, you can use the `@skip_doctest` decorator to indicate that a docstring should *not* be treated as a doctest at all. The difference between `# all-random` and `@skip_doctest` is that the former executes the example but ignores output, while the latter doesn't execute any code. `@skip_doctest` should be used for docstrings whose examples are purely informational.

If a given docstring fails under certain conditions but otherwise is a good doctest, you can use code like the following, that relies on the 'null' decorator to leave the docstring intact where it works as a test:

```
# The docstring for full_path doctests differently on win32 (different path
# separator) so just skip the doctest there, and use a null decorator
# elsewhere:

doctest_deco = dec.skip_doctest if sys.platform == 'win32' else dec.null_deco

@doctest_deco
def full_path(startPath, files):
    """Make full paths for all the listed files, based on startPath..."""

    # function body follows...
```


With our nose plugin that understands IPython syntax, an extremely effective way to write tests is to simply copy and paste an interactive session into a docstring. You can writing this type of test, where your docstring is meant *only* as a test, by prefixing the function name with `doctest_` and leaving its body *absolutely empty* other than the docstring. In `IPython.core.tests.test_magic` you can find several examples of this, but for completeness sake, your code should look like this (a simple case):

```
def doctest_time():
    """
    In [10]: %time None
    CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
    Wall time: 0.00 s
    """
```

This function is only analyzed for its docstring but it is not considered a separate test, which is why its body should be empty.

Parametric tests done right

If you need to run multiple tests inside the same standalone function or method of a `unittest.TestCase` subclass, IPython provides the `parametric` decorator for this purpose. This is superior to how test generators work in nose, because IPython's keeps intact your stack, which makes debugging vastly easier. For example, these are some parametric tests both in class form and as a standalone function (choose in each situation the style that best fits the problem at hand, since both work):

```
from IPython.testing import decorators as dec

def is_smaller(i,j):
    assert i<j,"%s !< %s" % (i,j)

class Tester(ParametricTestCase):

    def test_parametric(self):
        yield is_smaller(3, 4)
        x, y = 1, 2
        yield is_smaller(x, y)

@dec.parametric
def test_par_standalone():
    yield is_smaller(3, 4)
    x, y = 1, 2
    yield is_smaller(x, y)
```

Writing tests for Twisted-using code

Tests of Twisted [\[Twisted\]](#) using code should be written by subclassing the `TestCase` class that comes with `twisted.trial.unittest`. Furthermore, all `Deferred` instances that are created in the test must be properly chained and the final one *must* be the return value of the test method.

Note: The best place to see how to use the testing tools, are the tests for these tools themselves, which live in `IPython.testing.tests`.

7.5.4 Design requirements

This section is a set of notes on the key points of the IPython testing needs, that were used when writing the system and should be kept for reference as it evolves.

Testing IPython in full requires modifications to the default behavior of nose and doctest, because the IPython prompt is not recognized to determine Python input, and because IPython admits user input that is not valid Python (things like `%magics` and `!system` commands).

We basically need to be able to test the following types of code:

1. Pure Python files containing normal tests. These are not a problem, since Nose will pick them up as long as they conform to the (flexible) conventions used by nose to recognize tests.
2. Python files containing doctests. Here, we have two possibilities: - The prompts are the usual `>>>` and the input is pure Python. - The prompts are of the form `In [1]:` and the input can contain extended

IPython expressions.

In the first case, Nose will recognize the doctests as long as it is called with the `--with-doctest` flag. But the second case will likely require modifications or the writing of a new doctest plugin for Nose that is IPython-aware.

3. ReStructuredText files that contain code blocks. For this type of file, we have three distinct possibilities for the code blocks: - They use `>>>` prompts. - They use `In [1]:` prompts. - They are standalone blocks of pure Python code without any prompts.

The first two cases are similar to the situation #2 above, except that in this case the doctests must be extracted from input code blocks using docutils instead of from the Python docstrings.

In the third case, we must have a convention for distinguishing code blocks that are meant for execution from others that may be snippets of shell code or other examples not meant to be run. One possibility is to assume that all indented code blocks are meant for execution, but to have a special docutils directive for input that should not be executed.

For those code blocks that we will execute, the convention used will simply be that they get called and are considered successful if they run to completion without raising errors. This is similar to what Nose does for standalone test functions, and by putting asserts or other forms of exception-raising statements it becomes possible to have literate examples that double as lightweight tests.

4. Extension modules with doctests in function and method docstrings. Currently Nose simply can't find these docstrings correctly, because the underlying doctest DocTestFinder object fails there. Similarly to #2 above, the docstrings could have either pure python or IPython prompts.

Of these, only 3-c (reST with standalone code blocks) is not implemented at this point.

7.6 Releasing IPython

This section contains notes about the process that is used to release IPython. Our release process is currently not very formal and could be improved.

Most of the release process is automated by the `release` script in the `tools` directory. This is just a handy reminder for the release manager.

1. First, run `build_release`, which does all the file checking and building that the real release script will do. This will let you do test installations, check that the build procedure runs OK, etc. You may want to disable a few things like multi-version RPM building while testing, because otherwise the build takes really long.
2. Run the release script, which makes the tar.gz, eggs and Win32 .exe installer. It posts them to the site and registers the release with PyPI.
3. Update the website with announcements and links to the updated `changes.txt` in html form. Remember to put a short note both on the news page of the site and on Launchpad.
4. Drafting a short release announcement with i) highlights and ii) a link to the html version of the *Whats new* section of the documentation.
5. Make sure that the released version of the docs is live on the site.
6. Celebrate!

7.7 Development roadmap

IPython is an ambitious project that is still under heavy development. However, we want IPython to become useful to as many people as possible, as quickly as possible. To help us accomplish this, we are laying out a roadmap of where we are headed and what needs to happen to get there. Hopefully, this will help the IPython developers figure out the best things to work on for each upcoming release.

7.7.1 Work targeted to particular releases

Release 0.11

- Full module and package reorganization (done).
- Removal of the threaded shells and new implementation of GUI support based on `PyOSInputHook` (done).
- Refactor the configuration system (done).
- Prepare to refactor IPython's core by creating a new component and application system (done).
- Start to refactor IPython's core by turning everything into components (started).

Release 0.12

- Continue to refactor IPython's core by turning everything into components.

7.7.2 Major areas of work

Refactoring the main IPython core

During the summer of 2009, we began refactoring IPython's core. The main thrust in this work was to make the IPython core into a set of loosely coupled components. The base component class for this is `IPython.core.component.Component`. This section outlines the status of this work.

Parts of the IPython core that have been turned into components:

- The main `InteractiveShell` class.
- The aliases (`IPython.core.alias`).
- The `display` and `builtin` traps (`IPython.core.display_trap` and `IPython.core.builtin_trap`).
- The prefilter machinery (`IPython.core.prefilter`).

Parts of the IPythoncore that still need to be turned into components:

- Magics.
- Input and output history management.
- Prompts.
- Tab completers.
- Logging.
- Exception handling.
- Anything else.

Process management for `IPython.kernel`

A number of things need to be done to improve how processes are started up and managed for the parallel computing side of IPython:

- All of the processes need to use the new configuration system, components and application.
- We need to add support for other batch systems.

Performance problems

Currently, we have a number of performance issues in `IPython.kernel`:

- The controller stores a large amount of state in Python dictionaries. Under heavy usage, these dicts with get very large, causing memory usage problems. We need to develop more scalable solutions to this problem. This will also help the controller to be more fault tolerant.
- We currently don't have a good way of handling large objects in the controller. The biggest problem is that because we don't have any way of streaming objects, we get lots of temporary copies in the low-level buffers. We need to implement a better serialization approach and true streaming support.
- The controller currently unpickles and repickles objects. We need to use the `[push|pull]_serialized` methods instead.
- Currently the controller is a bottleneck. The best approach for this is to separate the controller itself into multiple processes, one for the core controller and one each for the controller interfaces.

7.7.3 Porting to 3.0

There are no definite plans for porting of IPython to Python 3. The major issue is the dependency on Twisted framework for the networking/threading stuff. It is possible that it the traditional IPython interactive console could be ported more easily since it has no such dependency. Here are a few things that will need to be considered when doing such a port especially if we want to have a codebase that works directly on both 2.x and 3.x.

1. The syntax for exceptions changed (PEP 3110). The old *except exc, var* changed to *except exc as var*. At last count there was 78 occurrences of this usage in the code base. This is a particularly problematic issue, because it's not easy to implement it in a 2.5-compatible way.

Because it is quite difficult to support simultaneously Python 2.5 and 3.x, we will likely at some point put out a release that requires strictly 2.6 and abandons 2.5 compatibility. This will then allow us to port the code to using `print()` as a function, *except exc as var* syntax, etc. But as of version 0.11 at least, we will retain Python 2.5 compatibility.

7.8 IPython module organization

As of the 0.11 release of IPython, the top-level packages and modules have been completely reorganized. This section describes the purpose of the top-level IPython subpackages.

7.8.1 Subpackage descriptions

- `IPython.config`. This package contains the configuration system of IPython, as well as default configuration files for the different IPython applications.
- `IPython.core`. This sub-package contains the core of the IPython interpreter, but none of its extended capabilities.
- `IPython.deathrow`. This is for code that is outdated, untested, rotting, or that belongs in a separate third party project. Eventually all this code will either i) be revived by someone willing to maintain it with tests and docs and re-included into IPython or 2) be removed from IPython proper, but put into a separate third-party Python package. No new code will be allowed here. If your favorite

extension has been moved here please contact the IPython developer mailing list to help us determine the best course of action.

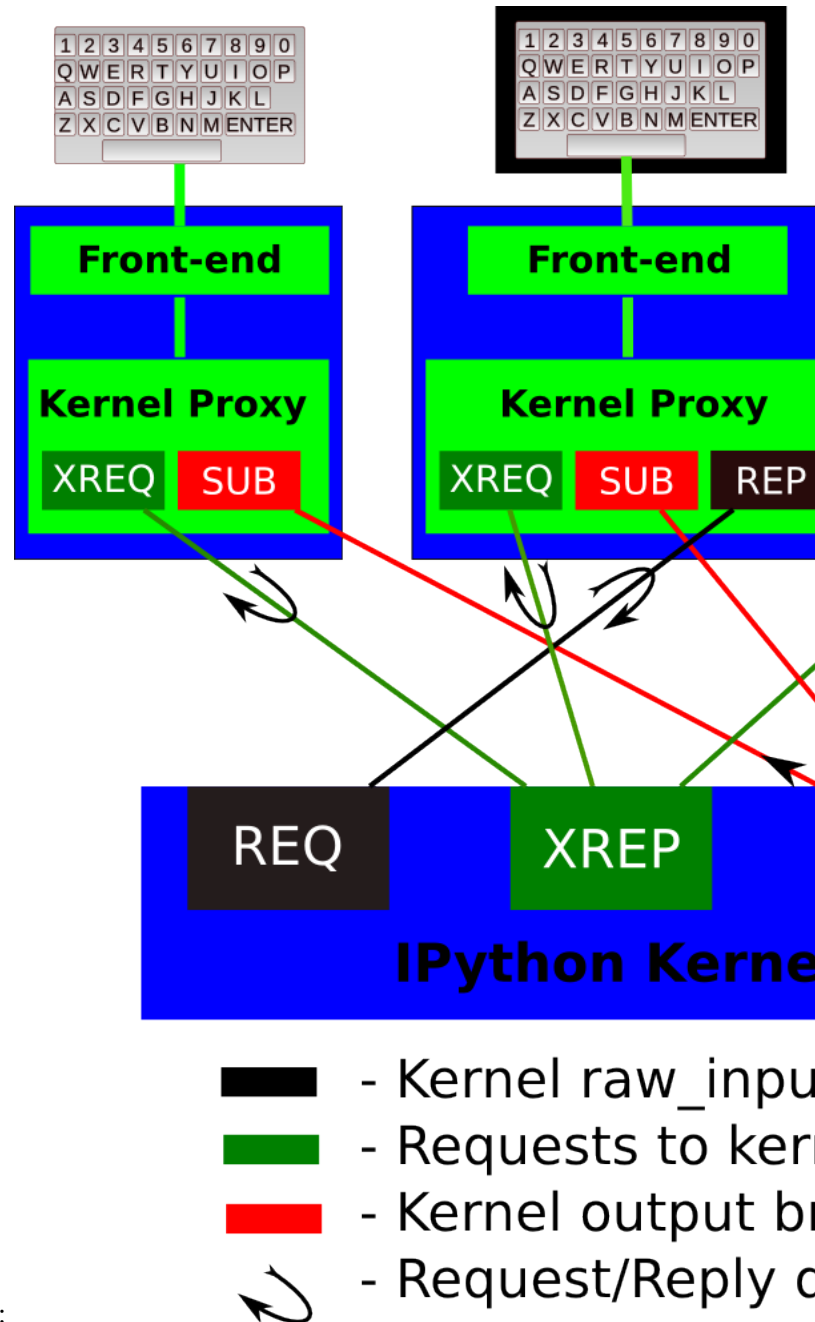
- `IPython.extensions`. This package contains fully supported IPython extensions. These extensions adhere to the official IPython extension API and can be enabled by adding them to a field in the configuration file. If your extension is no longer in this location, please look in `IPython.quarantine` and `IPython.deathrow` and contact the IPython developer mailing list.
- `IPython.external`. This package contains third party packages and modules that IPython ships internally to reduce the number of dependencies. Usually, these are short, single file modules.
- `IPython.frontend`. This package contains the various IPython frontends. Currently, the code in this subpackage is very experimental and may be broken.
- `IPython.gui`. Another semi-experimental wxPython based IPython GUI.
- `IPython.kernel`. This contains IPython's parallel computing system.
- `IPython.lib`. IPython has many extended capabilities that are not part of the IPython core. These things will go here and in. Modules in this package are similar to extensions, but don't adhere to the official IPython extension API.
- `IPython.quarantine`. This is for code that doesn't meet IPython's standards, but that we plan on keeping. To be moved out of this sub-package a module needs to have approval of the core IPython developers, tests and documentation. If your favorite extension has been moved here please contact the IPython developer mailing list to help us determine the best course of action.
- `IPython.scripts`. This package contains a variety of top-level command line scripts. Eventually, these should be moved to the `scripts` subdirectory of the appropriate IPython subpackage.
- `IPython.utils`. This sub-package will contain anything that might eventually be found in the Python standard library, like things in `genutils`. Each sub-module in this sub-package should contain functions and classes that serve a single purpose and that don't depend on things in the rest of IPython.

7.9 Messaging in IPython

7.9.1 Introduction

This document explains the basic communications design and messaging specification for how the various IPython objects interact over a network transport. The current implementation uses the [ZeroMQ](#) library for messaging within and between hosts.

Note: This document should be considered the authoritative description of the IPython messaging protocol, and all developers are strongly encouraged to keep it updated as the implementation evolves, so that we have a single common reference for all protocol details.



The basic design is explained in the following diagram:

A single kernel can be simultaneously connected to one or more frontends. The kernel has three sockets that serve the following functions:

1. REQ: this socket is connected to a *single* frontend at a time, and it allows the kernel to request input from a frontend when `raw_input()` is called. The frontend holding the matching REP socket acts as a ‘virtual keyboard’ for the kernel while this communication is happening (illustrated in the figure by the black outline around the central keyboard). In practice, frontends may display such kernel requests using a special input widget or otherwise indicating that the user is to type input for the kernel instead of normal commands in the frontend.
2. XREP: this single sockets allows multiple incoming connections from frontends, and this is the socket where requests for code execution, object information, prompts, etc. are made to the kernel by any

frontend. The communication on this socket is a sequence of request/reply actions from each frontend and the kernel.

3. PUB: this socket is the ‘broadcast channel’ where the kernel publishes all side effects (stdout, stderr, etc.) as well as the requests coming from any client over the XREP socket and its own requests on the REP socket. There are a number of actions in Python which generate side effects: `print()` writes to `sys.stdout`, errors generate tracebacks, etc. Additionally, in a multi-client scenario, we want all frontends to be able to know what each other has sent to the kernel (this can be useful in collaborative scenarios, for example). This socket allows both side effects and the information about communications taking place with one client over the XREQ/XREP channel to be made available to all clients in a uniform manner.

All messages are tagged with enough information (details below) for clients to know which messages come from their own interaction with the kernel and which ones are from other clients, so they can display each type appropriately.

The actual format of the messages allowed on each of these channels is specified below. Messages are dicts of dicts with string keys and values that are reasonably representable in JSON. Our current implementation uses JSON explicitly as its message format, but this shouldn’t be considered a permanent feature. As we’ve discovered that JSON has non-trivial performance issues due to excessive copying, we may in the future move to a pure pickle-based raw message format. However, it should be possible to easily convert from the raw objects to JSON, since we may have non-python clients (e.g. a web frontend). As long as it’s easy to make a JSON version of the objects that is a faithful representation of all the data, we can communicate with such clients.

Note: Not all of these have yet been fully fleshed out, but the key ones are, see kernel and frontend files for actual implementation details.

7.9.2 Python functional API

As messages are dicts, they map naturally to a `func(**kw)` call form. We should develop, at a few key points, functional forms of all the requests that take arguments in this manner and automatically construct the necessary dict for sending.

7.9.3 General Message Format

All messages send or received by any IPython process should have the following generic structure:

```
{
    # The message header contains a pair of unique identifiers for the
    # originating session and the actual message id, in addition to the
    # username for the process that generated the message. This is useful in
    # collaborative settings where multiple users may be interacting with the
    # same kernel simultaneously, so that frontends can label the various
    # messages in a meaningful way.
    'header' : { 'msg_id' : uuid,
                 'username' : str,
                 'session' : uuid
```



```
    },

    # In a chain of messages, the header from the parent is copied so that
    # clients can track where messages come from.
    'parent_header' : dict,

    # All recognized message type strings are listed below.
    'msg_type' : str,

    # The actual content of the message must be a dict, whose structure
    # depends on the message type.x
    'content' : dict,
}
```

For each message type, the actual content will differ and all existing message types are specified in what follows of this document.

7.9.4 Messages on the XREP/XREQ socket

Execute

This message type is used by frontends to ask the kernel to execute code on behalf of the user, in a namespace reserved to the user's variables (and thus separate from the kernel's own internal code and variables).

Message type: `execute_request`:

```
content = {
    # Source code to be executed by the kernel, one or more lines.
    'code' : str,

    # A boolean flag which, if True, signals the kernel to execute this
    # code as quietly as possible. This means that the kernel will compile
    # the code with IPython/core/tests/h 'exec' instead of 'single' (so
    # sys.displayhook will not fire), and will *not*:
    # - broadcast exceptions on the PUB socket
    # - do any logging
    # - populate any history
    #
    # The default is False.
    'silent' : bool,

    # A list of variable names from the user's namespace to be retrieved. What
    # returns is a JSON string of the variable's repr(), not a python object.
    'user_variables' : list,

    # Similarly, a dict mapping names to expressions to be evaluated in the
    # user's dict.
    'user_expressions' : dict,
}
```

The code field contains a single string (possibly multiline). The kernel is responsible for splitting this into one or more independent execution blocks and deciding whether to compile these in 'single' or 'exec' mode

(see below for detailed execution semantics).

The `user_` fields deserve a detailed explanation. In the past, IPython had the notion of a prompt string that allowed arbitrary code to be evaluated, and this was put to good use by many in creating prompts that displayed system status, path information, and even more esoteric uses like remote instrument status acquired over the network. But now that IPython has a clean separation between the kernel and the clients, the kernel has no prompt knowledge; prompts are a frontend-side feature, and it should be even possible for different frontends to display different prompts while interacting with the same kernel.

The kernel now provides the ability to retrieve data from the user's namespace after the execution of the main code, thanks to two fields in the `execute_request` message:

- `user_variables`: If only variables from the user's namespace are needed, a list of variable names can be passed and a dict with these names as keys and their `repr()` as values will be returned.
- `user_expressions`: For more complex expressions that require function evaluations, a dict can be provided with string keys and arbitrary python expressions as values. The return message will contain also a dict with the same keys and the `repr()` of the evaluated expressions as value.

With this information, frontends can display any status information they wish in the form that best suits each frontend (a status line, a popup, inline for a terminal, etc).

Note: In order to obtain the current execution counter for the purposes of displaying input prompts, frontends simply make an execution request with an empty code string and `silent=True`.

Execution semantics

When the silent flag is false, the execution of use code consists of the following phases (in silent mode, only the `code` field is executed):

1. Run the `pre_runcode_hook`.
2. Execute the `code` field, see below for details.
3. If #2 succeeds, compute `user_variables` and `user_expressions` are computed. This ensures that any error in the latter don't harm the main code execution.
4. Call any method registered with `register_post_execute()`.

Warning: The API for running code before/after the main code block is likely to change soon. Both the `pre_runcode_hook` and the `register_post_execute()` are susceptible to modification, as we find a consistent model for both.

To understand how the `code` field is executed, one must know that Python code can be compiled in one of three modes (controlled by the `mode` argument to the `compile()` builtin):

single Valid for a single interactive statement (though the source can contain multiple lines, such as a for loop). When compiled in this mode, the generated bytecode contains special instructions that trigger the calling of `sys.displayhook()` for any expression in the block that returns a value. This means that a single statement can actually produce multiple calls to `sys.displayhook()`, if for

example it contains a loop where each iteration computes an unassigned expression would generate 10 calls:

```
for i in range(10):
    i**2
```

exec An arbitrary amount of source code, this is how modules are compiled. `sys.displayhook()` is *never* implicitly called.

eval A single expression that returns a value. `sys.displayhook()` is *never* implicitly called.

The `code` field is split into individual blocks each of which is valid for execution in ‘single’ mode, and then:

- If there is only a single block: it is executed in ‘single’ mode.
- If there is more than one block:
 - if the last one is a single line long, run all but the last in ‘exec’ mode and the very last one in ‘single’ mode. This makes it easy to type simple expressions at the end to see computed values.
 - if the last one is no more than two lines long, run all but the last in ‘exec’ mode and the very last one in ‘single’ mode. This makes it easy to type simple expressions at the end to see computed values. - otherwise (last one is also multiline), run all in ‘exec’ mode
 - otherwise (last one is also multiline), run all in ‘exec’ mode as a single unit.

Any error in retrieving the `user_variables` or evaluating the `user_expressions` will result in a simple error message in the return fields of the form:

```
[ERROR] ExceptionType: Exception message
```

The user can simply send the same variable name or expression for evaluation to see a regular traceback.

Errors in any registered `post_execute` functions are also reported similarly, and the failing function is removed from the `post_execution` set so that it does not continue triggering failures.

Upon completion of the execution request, the kernel *always* sends a reply, with a status code indicating what happened and additional data depending on the outcome. See [below](#) for the possible return codes and associated data.

Execution counter (old prompt number)

The kernel has a single, monotonically increasing counter of all execution requests that are made with `silent=False`. This counter is used to populate the `In[n]`, `Out[n]` and `_n` variables, so clients will likely want to display it in some form to the user, which will typically (but not necessarily) be done in the prompts. The value of this counter will be returned as the `execution_count` field of all `execute_reply` messages.

Execution results

Message type: `execute_reply`:

```
content = {
    # One of: 'ok' OR 'error' OR 'abort'
    'status' : str,

    # The global kernel counter that increases by one with each non-silent
    # executed request. This will typically be used by clients to display
    # prompt numbers to the user. If the request was a silent one, this will
    # be the current value of the counter in the kernel.
    'execution_count' : int,
}
```

When status is 'ok', the following extra fields are present:

```
{
    # The execution payload is a dict with string keys that may have been
    # produced by the code being executed. It is retrieved by the kernel at
    # the end of the execution and sent back to the front end, which can take
    # action on it as needed. See main text for further details.
    'payload' : dict,

    # Results for the user_variables and user_expressions.
    'user_variables' : dict,
    'user_expressions' : dict,

    # The kernel will often transform the input provided to it. If the
    # '---->' transform had been applied, this is filled, otherwise it's the
    # empty string. So transformations like magics don't appear here, only
    # autocall ones.
    'transformed_code' : str,
}
```

Execution payloads

The notion of an 'execution payload' is different from a return value of a given set of code, which normally is just displayed on the pyout stream through the PUB socket. The idea of a payload is to allow special types of code, typically magics, to populate a data container in the IPython kernel that will be shipped back to the caller via this channel. The kernel will have an API for this, probably something along the lines of:

```
ip.exec_payload_add(key, value)
```

though this API is still in the design stages. The data returned in this payload will allow frontends to present special views of what just happened.

When status is 'error', the following extra fields are present:

```
{
    'exc_name' : str,    # Exception name, as a string
    'exc_value' : str,  # Exception value, as a string

    # The traceback will contain a list of frames, represented each as a
    # string. For now we'll stick to the existing design of ultraTB, which
    # controls exception level of detail statefully. But eventually we'll
```

```

# want to grow into a model where more information is collected and
# packed into the traceback object, with clients deciding how little or
# how much of it to unpack. But for now, let's start with a simple list
# of strings, since that requires only minimal changes to ultratb as
# written.
'traceback' : list,
}

```

When status is 'abort', there are for now no additional data fields. This happens when the kernel was interrupted by a signal.

Kernel attribute access

Warning: This part of the messaging spec is not actually implemented in the kernel yet.

While this protocol does not specify full RPC access to arbitrary methods of the kernel object, the kernel does allow read (and in some cases write) access to certain attributes.

The policy for which attributes can be read is: any attribute of the kernel, or its sub-objects, that belongs to a `Configurable` object and has been declared at the class-level with Traits validation, is in principle accessible as long as its name does not begin with a leading underscore. The attribute itself will have metadata indicating whether it allows remote read and/or write access. The message spec follows for attribute read and write requests.

Message type: `getattr_request`:

```

content = {
    # The (possibly dotted) name of the attribute
    'name' : str,
}

```

When a `getattr_request` fails, there are two possible error types:

- `AttributeError`: this type of error was raised when trying to access the given name by the kernel itself. This means that the attribute likely doesn't exist.
- `AccessError`: the attribute exists but its value is not readable remotely.

Message type: `getattr_reply`:

```

content = {
    # One of ['ok', 'AttributeError', 'AccessError'].
    'status' : str,
    # If status is 'ok', a JSON object.
    'value' : object,
}

```

Message type: `setattr_request`:

```

content = {
    # The (possibly dotted) name of the attribute
    'name' : str,
}

```

```
# A JSON-encoded object, that will be validated by the Traits  
# information in the kernel  
'value' : object,  
}
```

When a `setattr_request` fails, there are also two possible error types with similar meanings as those of the `getattr_request` case, but for writing.

Message type: `setattr_reply`:

```
content = {  
    # One of ['ok', 'AttributeError', 'AccessError'].  
    'status' : str,  
}
```

Object information

One of IPython's most used capabilities is the introspection of Python objects in the user's namespace, typically invoked via the `?` and `??` characters (which in reality are shorthands for the `%pinfo` magic). This is used often enough that it warrants an explicit message type, especially because frontends may want to get object information in response to user keystrokes (like Tab or F1) besides from the user explicitly typing code like `x??`.

Message type: `object_info_request`:

```
content = {  
    # The (possibly dotted) name of the object to be searched in all  
    # relevant namespaces  
    'name' : str,  
  
    # The level of detail desired. The default (0) is equivalent to typing  
    # 'x?' at the prompt, 1 is equivalent to 'x??'.  
    'detail_level' : int,  
}
```

The returned information will be a dictionary with keys very similar to the field names that IPython prints at the terminal.

Message type: `object_info_reply`:

```
content = {  
    # The name the object was requested under  
    'name' : str,  
  
    # Boolean flag indicating whether the named object was found or not. If  
    # it's false, all other fields will be empty.  
    'found' : bool,  
  
    # Flags for magics and system aliases  
    'ismagic' : bool,  
    'isalias' : bool,
```

```
# The name of the namespace where the object was found ('builtin',
# 'magics', 'alias', 'interactive', etc.)
'namespace' : str,

# The type name will be type.__name__ for normal Python objects, but it
# can also be a string like 'Magic function' or 'System alias'
'type_name' : str,

'string_form' : str,

# For objects with a __class__ attribute this will be set
'base_class' : str,

# For objects with a __len__ attribute this will be set
'length' : int,

# If the object is a function, class or method whose file we can find,
# we give its full path
'file' : str,

# For pure Python callable objects, we can reconstruct the object
# definition line which provides its call signature. For convenience this
# is returned as a single 'definition' field, but below the raw parts that
# compose it are also returned as the argspec field.
'definition' : str,

# The individual parts that together form the definition string. Clients
# with rich display capabilities may use this to provide a richer and more
# precise representation of the definition line (e.g. by highlighting
# arguments based on the user's cursor position). For non-callable
# objects, this field is empty.
'argspec' : { # The names of all the arguments
    args : list,
    # The name of the varargs (*args), if any
    varargs : str,
    # The name of the varkw (**kw), if any
    varkw : str,
    # The values (as strings) of all default arguments. Note
    # that these must be matched *in reverse* with the 'args'
    # list above, since the first positional args have no default
    # value at all.
    defaults : list,
},

# For instances, provide the constructor signature (the definition of
# the __init__ method):
'init_definition' : str,

# Docstrings: for any object (function, method, module, package) with a
# docstring, we show it. But in addition, we may provide additional
# docstrings. For example, for instances we will show the constructor
# and class docstrings as well, if available.
'docstring' : str,
```

```
# For instances, provide the constructor and class docstrings
'init_docstring' : str,
'class_docstring' : str,

# If it's a callable object whose call method has a separate docstring and
# definition line:
'call_def' : str,
'call_docstring' : str,

# If detail_level was 1, we also try to find the source code that
# defines the object, if possible. The string 'None' will indicate
# that no source was found.
'source' : str,
}

,
```

Complete

Message type: complete_request:

```
content = {
    # The text to be completed, such as 'a.is'
    'text' : str,

    # The full line, such as 'print a.is'. This allows completers to
    # make decisions that may require information about more than just the
    # current word.
    'line' : str,

    # The entire block of text where the line is. This may be useful in the
    # case of multiline completions where more context may be needed. Note: if
    # in practice this field proves unnecessary, remove it to lighten the
    # messages.

    'block' : str,

    # The position of the cursor where the user hit 'TAB' on the line.
    'cursor_pos' : int,
}
```

Message type: complete_reply:

```
content = {
    # The list of all matches to the completion request, such as
    # ['a.isalnum', 'a.isalpha'] for the above example.
    'matches' : list
}
```


History

For clients to explicitly request history from a kernel. The kernel has all the actual execution history stored in a single location, so clients can request it from the kernel when needed.

Message type: `history_request`:

```
content = {  
  
    # If True, also return output history in the resulting dict.  
    'output' : bool,  
  
    # If True, return the raw input history, else the transformed input.  
    'raw' : bool,  
  
    # This parameter can be one of: A number, a pair of numbers, None  
    # If not given, last 40 are returned.  
    # - number n: return the last n entries.  
    # - pair n1, n2: return entries in the range(n1, n2).  
    # - None: return all history  
    'index' : n or (n1, n2) or None,  
}
```

Message type: `history_reply`:

```
content = {  
    # A dict with prompt numbers as keys and either (input, output) or input  
    # as the value depending on whether output was True or False,  
    # respectively.  
    'history' : dict,  
}
```

Connect

When a client connects to the request/reply socket of the kernel, it can issue a connect request to get basic information about the kernel, such as the ports the other ZeroMQ sockets are listening on. This allows clients to only have to know about a single port (the XREQ/XREP channel) to connect to a kernel.

Message type: `connect_request`:

```
content = {  
}
```

Message type: `connect_reply`:

```
content = {  
    'xrep_port' : int    # The port the XREP socket is listening on.  
    'pub_port'  : int    # The port the PUB socket is listening on.  
    'req_port'  : int    # The port the REQ socket is listening on.  
    'hb_port'   : int    # The port the heartbeat socket is listening on.  
}
```

Kernel shutdown

The clients can request the kernel to shut itself down; this is used in multiple cases:

- when the user chooses to close the client application via a menu or window control.
- when the user types ‘exit’ or ‘quit’ (or their uppercase magic equivalents).
- when the user chooses a GUI method (like the ‘Ctrl-C’ shortcut in the IPythonQt client) to force a kernel restart to get a clean kernel without losing client-side state like history or inlined figures.

The client sends a shutdown request to the kernel, and once it receives the reply message (which is otherwise empty), it can assume that the kernel has completed shutdown safely.

Upon their own shutdown, client applications will typically execute a last minute sanity check and forcefully terminate any kernel that is still alive, to avoid leaving stray processes in the user’s machine.

For both shutdown request and reply, there is no actual content that needs to be sent, so the content dict is empty.

Message type: shutdown_request:

```
content = {
    'restart' : bool # whether the shutdown is final, or precedes a restart
}
```

Message type: shutdown_reply:

```
content = {
    'restart' : bool # whether the shutdown is final, or precedes a restart
}
```

Note: When the clients detect a dead kernel thanks to inactivity on the heartbeat socket, they simply send a forceful process termination signal, since a dead process is unlikely to respond in any useful way to messages.

7.9.5 Messages on the PUB/SUB socket

Streams (stdout, stderr, etc)

Message type: stream:

```
content = {
    # The name of the stream is one of 'stdin', 'stdout', 'stderr'
    'name' : str,

    # The data is an arbitrary string to be written to that stream
    'data' : str,
}
```

When a kernel receives a `raw_input` call, it should also broadcast it on the pub socket with the names `'stdin'` and `'stdin_reply'`. This will allow other clients to monitor/display kernel interactions and possibly replay them to their user or otherwise expose them.

Display Data

This type of message is used to bring back data that should be displayed (text, html, svg, etc.) in the frontends. This data is published to all frontends. Each message can have multiple representations of the data; it is up to the frontend to decide which to use and how. A single message should contain all possible representations of the same information. Each representation should be a JSON'able data structure, and should be a valid MIME type.

Some questions remain about this design:

- Do we use this message type for `pyout/displayhook`? Probably not, because the `displayhook` also has to handle the Out prompt display. On the other hand we could put that information into the metadata section.

Message type: `display_data`:

```
content = {  
  
    # Who create the data  
    'source' : str,  
  
    # The data dict contains key/value pairs, where the keys are MIME  
    # types and the values are the raw data of the representation in that  
    # format. The data dict must minimally contain the 'text/plain'  
    # MIME type which is used as a backup representation.  
    'data' : dict,  
  
    # Any metadata that describes the data  
    'metadata' : dict  
}
```

Python inputs

These messages are the re-broadcast of the `execute_request`.

Message type: `pyin`:

```
content = {  
    'code' : str # Source code to be executed, one or more lines  
}
```

Python outputs

When Python produces output from code that has been compiled in with the `'single'` flag to `compile()`, any expression that produces a value (such as `1+1`) is passed to `sys.displayhook`, which is a callable that can do with this value whatever it wants. The default behavior of `sys.displayhook` in the Python

interactive prompt is to print to `sys.stdout` the `repr()` of the value as long as it is not `None` (which isn't printed at all). In our case, the kernel instantiates as `sys.displayhook` an object which has similar behavior, but which instead of printing to `stdout`, broadcasts these values as `pyout` messages for clients to display appropriately.

IPython's `displayhook` can handle multiple simultaneous formats depending on its configuration. The default pretty-printed `repr` text is always given with the `data` entry in this message. Any other formats are provided in the `extra_formats` list. Frontends are free to display any or all of these according to its capabilities. `extra_formats` list contains 3-tuples of an ID string, a type string, and the data. The ID is unique to the formatter implementation that created the data. Frontends will typically ignore the ID unless it has requested a particular formatter. The type string tells the frontend how to interpret the data. It is often, but not always a MIME type. Frontends should ignore types that it does not understand. The data itself is any JSON object and depends on the format. It is often, but not always a string.

Message type: `pyout`:

```
content = {  
  
    # The counter for this execution is also provided so that clients can  
    # display it, since IPython automatically creates variables called _N  
    # (for prompt N).  
    'execution_count' : int,  
  
    # The data dict contains key/value pairs, where the keys are MIME  
    # types and the values are the raw data of the representation in that  
    # format. The data dict must minimally contain the ``text/plain``  
    # MIME type which is used as a backup representation.  
    'data' : dict,  
  
}
```

Python errors

When an error occurs during code execution

Message type: `pyerr`:

```
content = {  
    # Similar content to the execute_reply messages for the 'error' case,  
    # except the 'status' field is omitted.  
}
```

Kernel status

This message type is used by frontends to monitor the status of the kernel.

Message type: `status`:

```
content = {  
    # When the kernel starts to execute code, it will enter the 'busy'  
    # state and when it finishes, it will enter the 'idle' state.
```

```
    execution_state : ('busy', 'idle')
}
```

Kernel crashes

When the kernel has an unexpected exception, caught by the last-resort `sys.excepthook`, we should broadcast the crash handler's output before exiting. This will allow clients to notice that a kernel died, inform the user and propose further actions.

Message type: `crash`:

```
content = {
    # Similarly to the 'error' case for execute_reply messages, this will
    # contain exc_name, exc_type and traceback fields.

    # An additional field with supplementary information such as where to
    # send the crash message
    'info' : str,
}
```

Future ideas

Other potential message types, currently unimplemented, listed below as ideas.

Message type: `file`:

```
content = {
    'path' : 'cool.jpg',
    'mimetype' : str,
    'data' : str,
}
```

7.9.6 Messages on the REQ/REP socket

This is a socket that goes in the opposite direction: from the kernel to a *single* frontend, and its purpose is to allow `raw_input` and similar operations that read from `sys.stdin` on the kernel to be fulfilled by the client. For now we will keep these messages as simple as possible, since they basically only mean to convey the `raw_input(prompt)` call.

Message type: `input_request`:

```
content = { 'prompt' : str }
```

Message type: `input_reply`:

```
content = { 'value' : str }
```

Note: We do not explicitly try to forward the raw `sys.stdin` object, because in practice the kernel should behave like an interactive program. When a program is opened on the console, the keyboard effectively

takes over the `stdin` file descriptor, and it can't be used for raw reading anymore. Since the IPython kernel effectively behaves like a console program (albeit one whose “keyboard” is actually living in a separate process and transported over the zmq connection), raw `stdin` isn't expected to be available.

7.9.7 Heartbeat for kernels

Initially we had considered using messages like those above over ZMQ for a kernel ‘heartbeat’ (a way to detect quickly and reliably whether a kernel is alive at all, even if it may be busy executing user code). But this has the problem that if the kernel is locked inside extension code, it wouldn't execute the python heartbeat code. But it turns out that we can implement a basic heartbeat with pure ZMQ, without using any Python messaging at all.

The monitor sends out a single zmq message (right now, it is a str of the monitor's lifetime in seconds), and gets the same message right back, prefixed with the zmq identity of the XREQ socket in the heartbeat process. This can be a uuid, or even a full message, but there doesn't seem to be a need for packing up a message when the sender and receiver are the exact same Python object.

The model is this:

```
monitor.send(str(self.lifetime)) # '1.2345678910'
```

and the monitor receives some number of messages of the form:

```
['uuid-abcd-dead-beef', '1.2345678910']
```

where the first part is the zmq.IDENTITY of the heart's XREQ on the engine, and the rest is the message sent by the monitor. No Python code ever has any access to the message between the monitor's send, and the monitor's recv.

7.9.8 ToDo

Missing things include:

- Important: finish thinking through the payload concept and API.
- Important: ensure that we have a good solution for magics like `%edit`. It's likely that with the payload concept we can build a full solution, but not 100% clear yet.
- Finishing the details of the heartbeat protocol.
- Signal handling: specify what kind of information kernel should broadcast (or not) when it receives signals.

7.10 Messaging for Parallel Computing

This is an extension of the *messaging* doc. Diagrams of the connections can be found in the *parallel connections* doc.

ZMQ messaging is also used in the parallel computing IPython system. All messages to/from kernels remain the same as the single kernel model, and are forwarded through a ZMQ Queue device. The controller receives all messages and replies in these channels, and saves results for future use.

7.10.1 The Controller

The controller is the central process of the IPython parallel computing model. It has 3 Devices:

- Heartbeater
- Multiplexed Queue
- Task Queue

and 3 sockets:

- XREP for both engine and client registration
- PUB for notification of engine changes
- XREP for client requests

Registration (XREP)

The first function of the Controller is to facilitate and monitor connections of clients and engines. Both client and engine registration are handled by the same socket, so only one ip/port pair is needed to connect any number of connections and clients.

Engines register with the `zmq.IDENTITY` of their two XREQ sockets, one for the queue, which receives execute requests, and one for the heartbeat, which is used to monitor the survival of the Engine process.

Message type: `registration_request`:

```
content = {
    'queue'      : 'abcd-1234-...', # the queue XREQ id
    'heartbeat'  : '1234-abcd-...' # the heartbeat XREQ id
}
```

The Controller replies to an Engine's registration request with the engine's integer ID, and all the remaining connection information for connecting the heartbeat process, and kernel socket(s). The message status will be an error if the Engine requests IDs that already in use.

Message type: `registration_reply`:

```
content = {
    'status' : 'ok', # or 'error'
    # if ok:
    'id' : 0, # int, the engine id
    'queue' : 'tcp://127.0.0.1:12345', # connection for engine side of the queue
    'heartbeat' : (a,b), # tuple containing two interfaces needed for heartbeat
    'task' : 'tcp...', # addr for task queue, or None if no task queue running
    # if error:
    'reason' : 'queue_id already registered'
}
```

Clients use the same socket to start their connections. Connection requests from clients need no information:

Message type: `connection_request`:

```
content = {}
```

The reply to a Client registration request contains the connection information for the multiplexer and load balanced queues, as well as the address for direct controller queries. If any of these addresses is *None*, that functionality is not available.

Message type: `connection_reply`:

```
content = {
    'status' : 'ok', # or 'error'
    # if ok:
    'queue' : 'tcp://127.0.0.1:12345', # connection for client side of the queue
    'task' : 'tcp...', # addr for task queue, or None if no task queue running
    'controller' : 'tcp...' # addr for controller methods, like queue_request, etc.
}
```

Heartbeat

The controller uses a heartbeat system to monitor engines, and track when they become unresponsive. As described in *messages*, and shown in *connections*.

Notification (PUB)

The controller published all engine registration/unregistration events on a PUB socket. This allows clients to have up-to-date engine ID sets without polling. Registration notifications contain both the integer engine ID and the queue ID, which is necessary for sending messages via the Multiplexer Queue.

Message type: `registration_notification`:

```
content = {
    'id' : 0, # engine ID that has been registered
    'queue' : 'engine_id' # the IDENT for the engine's queue
}
```

Message type: `unregistration_notification`:

```
content = {
    'id' : 0 # engine ID that has been unregistered
}
```

Client Queries (XREP)

The controller monitors and logs all queue traffic, so that clients can retrieve past results or monitor pending tasks. Currently, this information resides in memory on the Controller, but will ultimately be offloaded to a database over an additional ZMQ connection. The interface should remain the same or at least similar.

`queue_request()` requests can specify multiple engines to query via the *targets* element. A *verbose* flag can be passed, to determine whether the result should be the list of *msg_ids* in the queue or simply the length of each list.

Message type: `queue_request`:

```
content = {
    'verbose' : True, # whether return should be lists themselves or just lens
    'targets' : [0,3,1] # list of ints
}
```

The content of a reply to a `:func:queue_request` request is a dict, keyed by the engine IDs. Note that they will be the string representation of the integer keys, since JSON cannot handle number keys.

Message type: `queue_reply`:

```
content = {
    '0' : {'completed' : 1, 'queue' : 7},
    '1' : {'completed' : 10, 'queue' : 1}
}
```

Clients can request individual results directly from the controller. This is primarily for use gathering results of executions not submitted by the particular client, as the client will have all its own results already. Requests are made by *msg_id*, and can contain one or more *msg_id*.

Message type: `result_request`:

```
content = {
    'msg_ids' : [uuid,'...'] # list of strs
}
```

The `result_request()` reply contains the content objects of the actual execution reply messages

Message type: `result_reply`:

```
content = {
    'status' : 'ok', # else error
    # if ok:
    msg_id : msg, # the content dict is keyed by msg_ids,
                # values are the result messages
    'pending' : ['msg_id','...'], # msg_ids still pending
    # if error:
    'reason' : "explanation"
}
```

Clients can also instruct the controller to forget the results of messages. This can be done by message ID or engine ID. Individual messages are dropped by *msg_id*, and all messages completed on an engine are dropped by engine ID.

If the *msg_ids* element is the string `'all'` instead of a list, then all completed results are forgotten.

Message type: `purge_request`:

```
content = {
    'msg_ids' : ['id1', 'id2',...], # list of msg_ids or 'all'
}
```

```
    'engine_ids' : [0,2,4] # list of engine IDs
}
```

The reply to a purge request is simply the status 'ok' if the request succeeded, or an explanation of why it failed, such as requesting the purge of a nonexistent or pending message.

Message type: `purge_reply`:

```
content = {
    'status' : 'ok', # or 'error'

    # if error:
    'reason' : "KeyError: no such msg_id 'whoda'"
}
```

`apply()` and `apply_bound()`

The `Namespace` model suggests that execution be able to use the model:

```
client.apply(f, *args, **kwargs)
```

which takes *f*, a function in the user's namespace, and executes `f(*args, **kwargs)` on a remote engine, returning the result (or, for non-blocking, information facilitating later retrieval of the result). This model, unlike the `execute` message which just uses a code string, must be able to send arbitrary (pickleable) Python objects. And ideally, copy as little data as we can. The *buffers* property of a `Message` was introduced for this purpose.

Utility method `build_apply_message()` in `IPython.zmq.streamsession` wraps a function signature and builds the correct buffer format.

Message type: `apply_request`:

```
content = {
    'bound' : True # whether to execute in the engine's namespace or unbound
}
buffers = ['...'] # at least 3 in length
                # as built by build_apply_message(f,args,kwargs)
```

Message type: `apply_reply`:

```
content = {
    'status' : 'ok' # 'ok' or 'error'
    # other error info here, as in other messages
}
buffers = ['...'] # either 1 or 2 in length
                # a serialization of the return value of f(*args,**kwargs)
                # only populated if status is 'ok'
```

7.10.2 Implementation

There are a few differences in implementation between the *StreamSession* object used in the parallel computing fork and the *Session* object, the main one being that messages are sent in parts, rather than as a single serialized object. *StreamSession* objects also take pack/unpack functions, which are to be used when serializing/deserializing objects. These can be any functions that translate to/from formats that ZMQ sockets can send (buffers, bytes, etc.).

Split Sends

Previously, messages were bundled as a single json object and one call to `socket.send_json()`. Since the controller inspects all messages, and doesn't need to see the content of the messages, which can be large, messages are serialized and sent in pieces. All messages are sent in at least 3 parts: the header, the parent header, and the content. This allows the controller to unpack and inspect the (always small) header, without spending time unpacking the content unless the message is bound for the controller. Buffers are added on to the end of the message, and can be any objects that present the buffer interface.

7.11 Connection Diagrams of The IPython ZMQ Cluster

This is a quick summary and illustration of the connections involved in the ZeroMQ based IPython cluster for parallel computing.

7.11.1 All Connections

The Parallel Computing code is currently under development in Min RK's IPython [fork](#) on GitHub.

The IPython cluster consists of a Controller and one or more clients and engines. The goal of the Controller is to manage and monitor the connections and communications between the clients and the engines.

It is important for security/practicality reasons that all connections be inbound to the controller process. The arrows in the figures indicate the direction of the connection.

The Controller consists of two ZMQ Devices - both MonitoredQueues, one for Tasks (load balanced, engine agnostic), one for Multiplexing (explicit targets), a Python device for monitoring (the Heartbeat Monitor).

Registration

Once a controller is launched, the only information needed for connecting clients and/or engines to the controller is the IP/port of the XREP socket called the Registrar. This socket handles connections from both clients and engines, and replies with the remaining information necessary to establish the remaining connections.

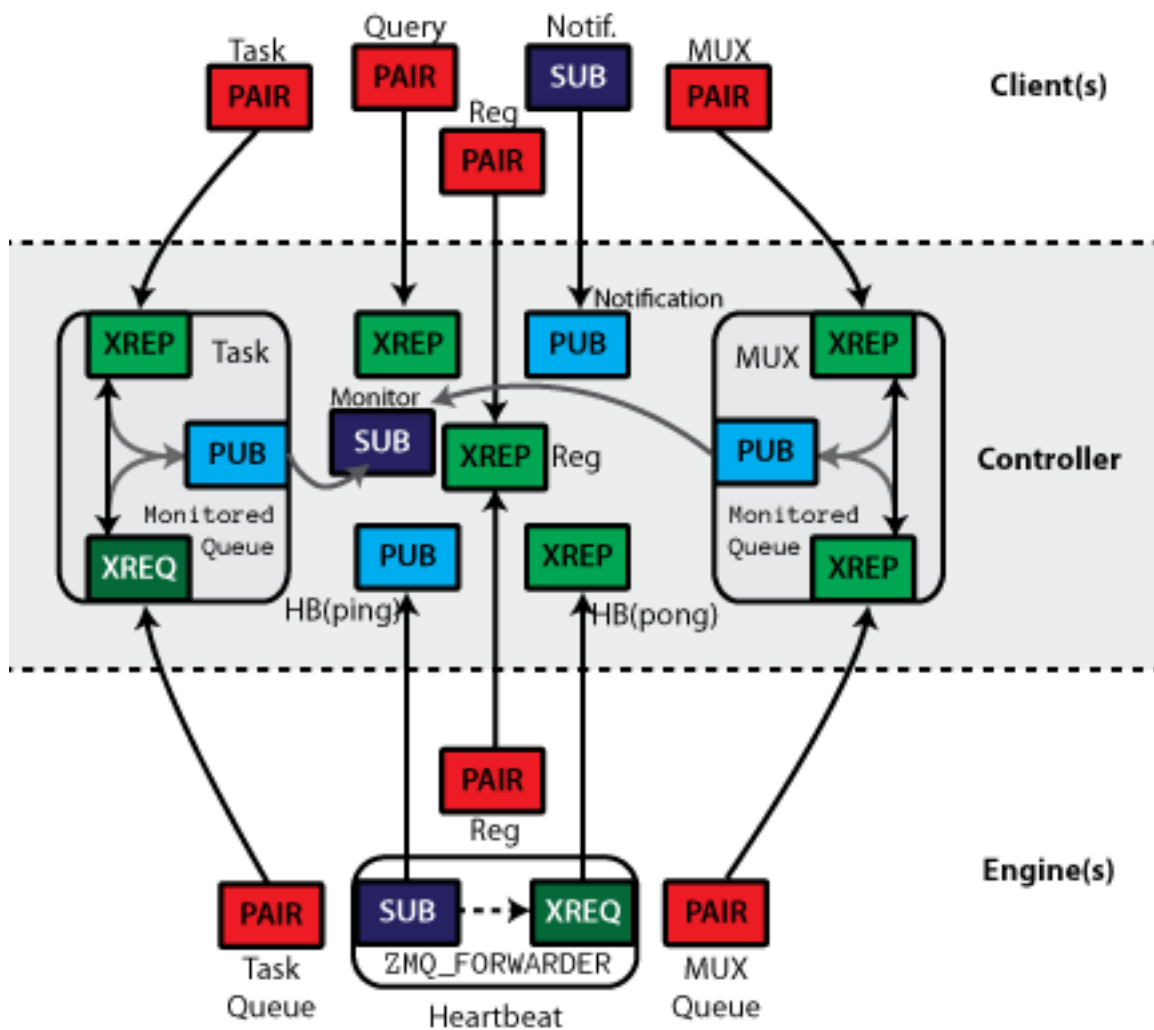


Figure 7.1: All the connections involved in connecting one client to one engine.

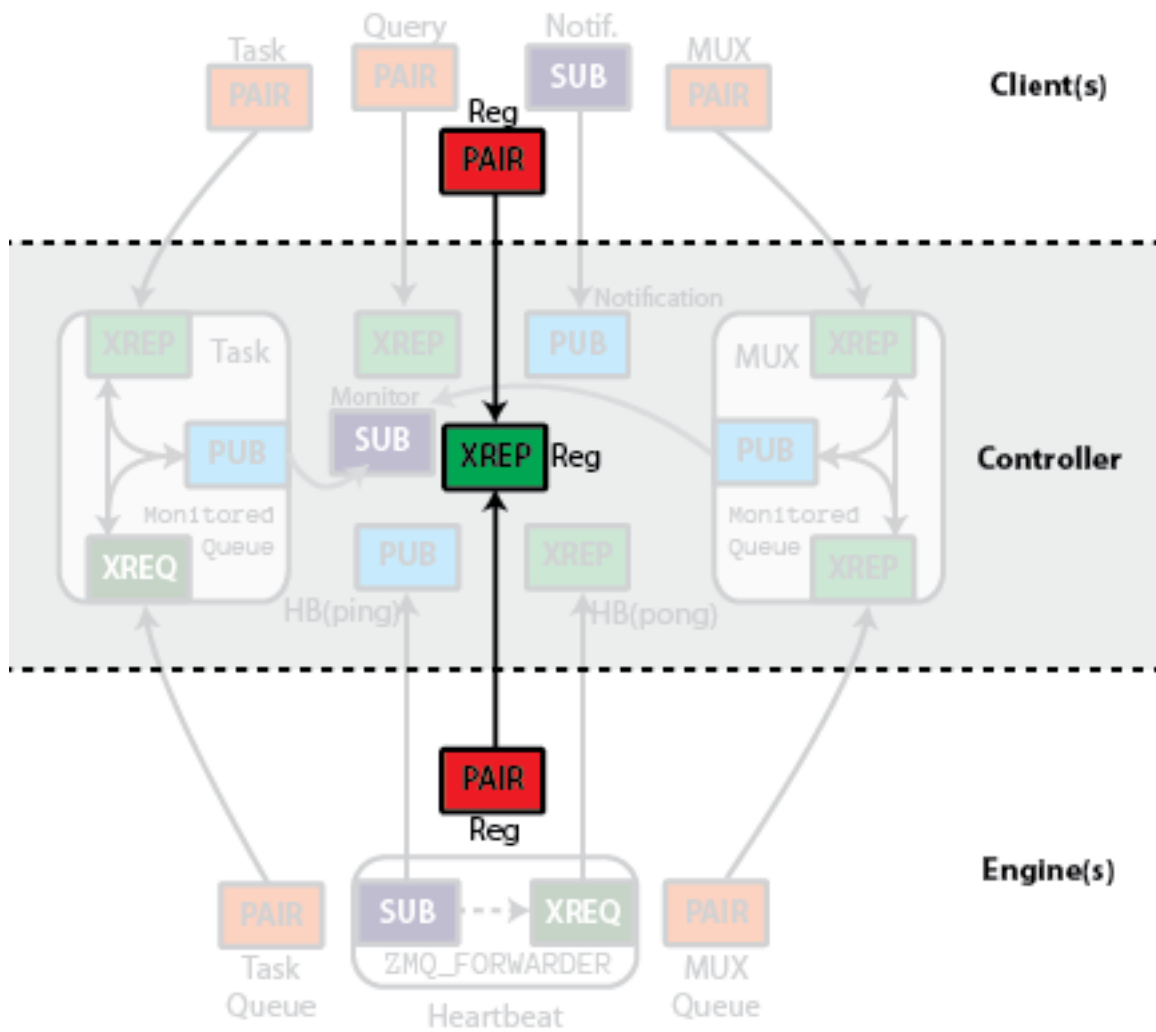


Figure 7.2: Engines and Clients only need to know where the Registrar XREP is located to start connecting.

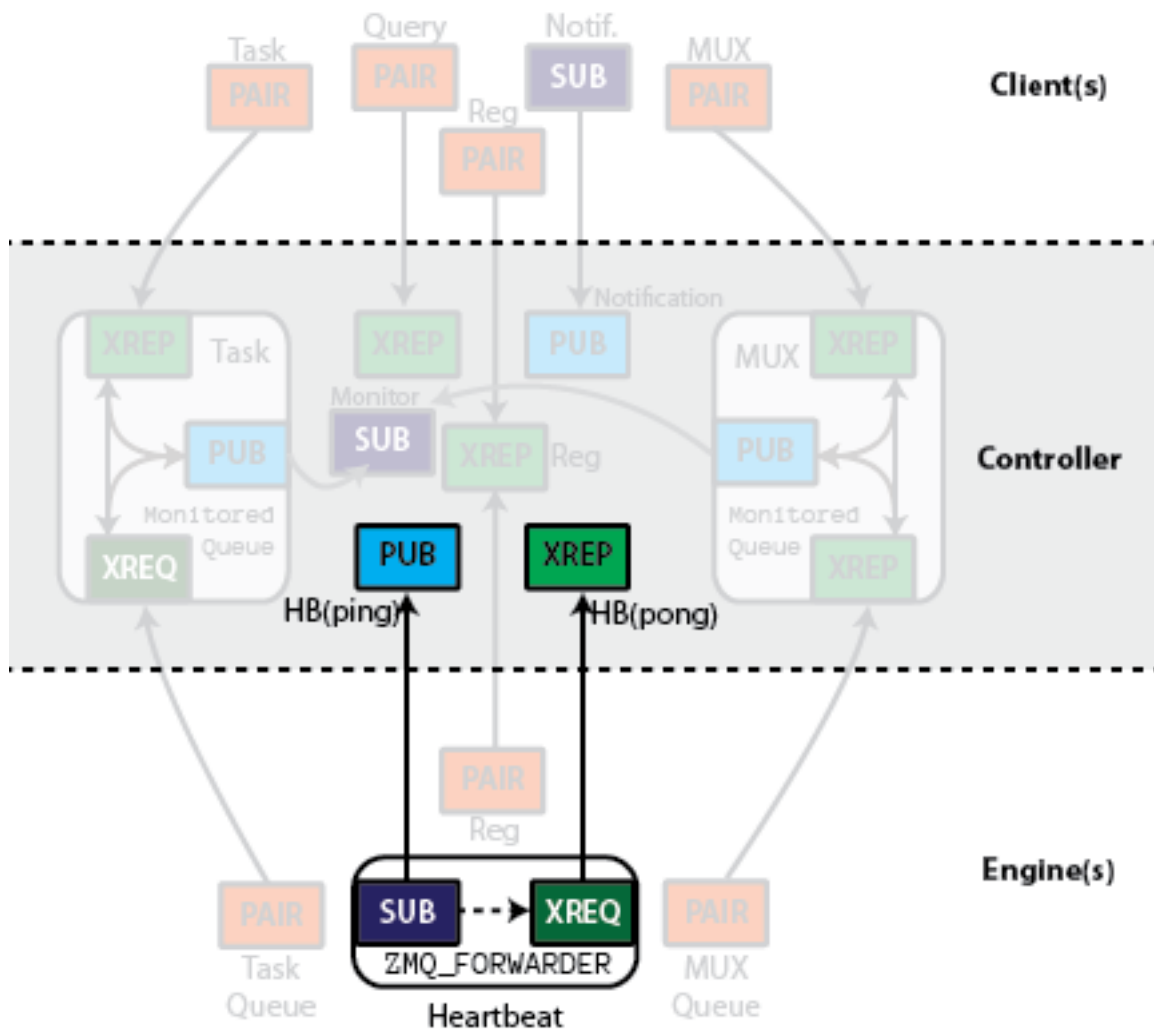


Figure 7.3: The heartbeat sockets.

Heartbeat

The heartbeat process has been described elsewhere. To summarize: the controller publishes a distinct message periodically via a PUB socket. Each engine has a `zmq.FORWARDER` device with a SUB socket for input, and XREQ socket for output. The SUB socket is connected to the PUB socket labeled *HB(ping)*, and the XREQ is connected to the XREP labeled *HB(pong)*. This results in the same message being relayed back to the Heartbeat Monitor with the addition of the XREQ prefix. The Heartbeat Monitor receives all the replies via an XREP socket, and identifies which hearts are still beating by the `zmq.IDENTITY` prefix of the XREQ sockets.

Queues

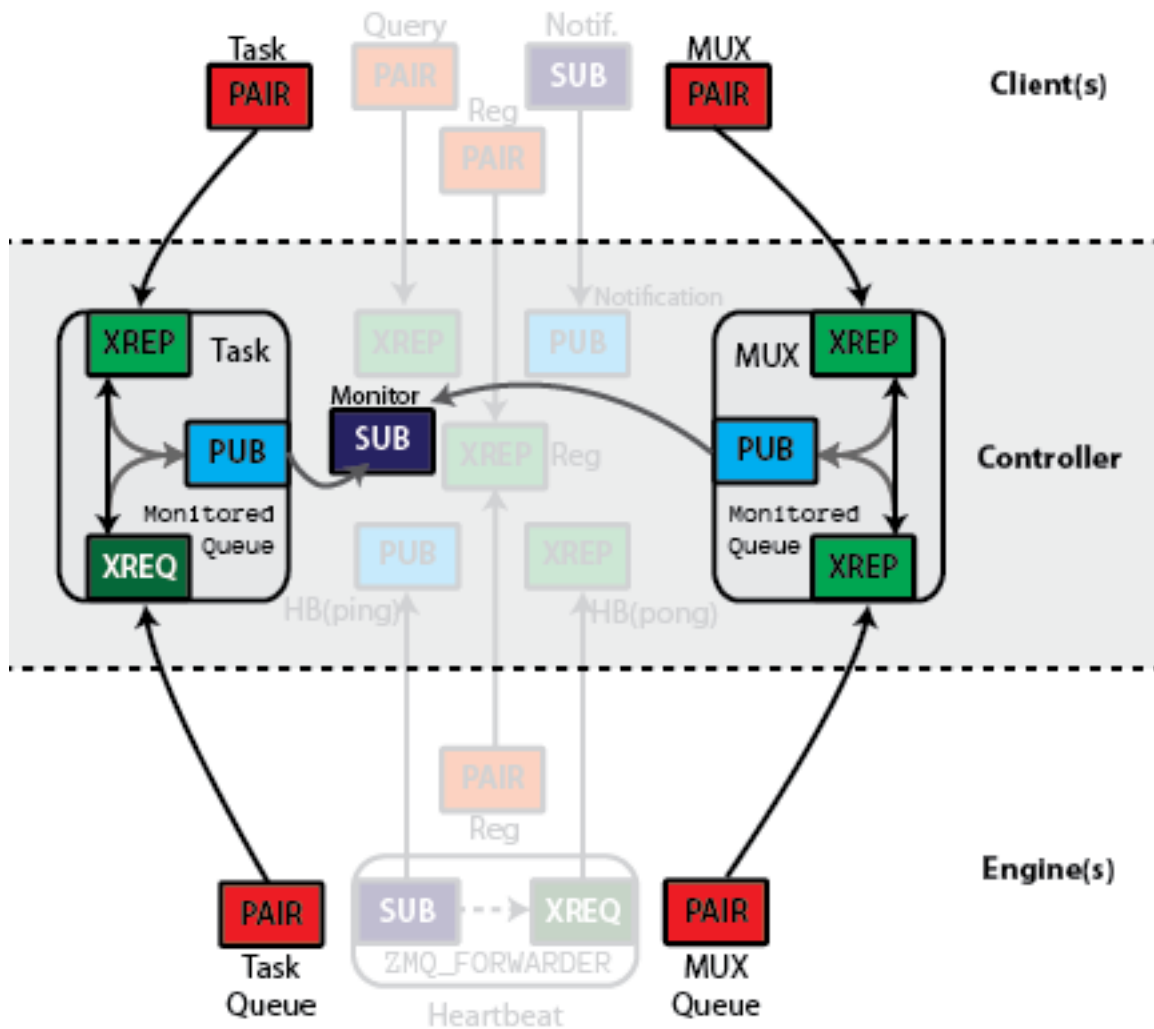


Figure 7.4: Load balanced Task queue on the left, explicitly multiplexed queue on the right.

The controller has two `MonitoredQueue` devices. These devices are primarily for relaying messages between clients and engines, but the controller needs to see those messages for its own purposes. Since no Python code may exist between the two sockets in a queue, all messages sent through these queues (both directions) are also sent via a `PUB` socket to a monitor, which allows the Controller to monitor queue traffic without interfering with it.

For tasks, the engine need not be specified. Messages sent to the `XREP` socket from the client side are assigned to an engine via `ZMQ`'s `XREQ` round-robin load balancing. Engine replies are directed to specific clients via the `IDENTITY` of the client, which is received as a prefix at the Engine.

For Multiplexing, `XREP` is used for both in and output sockets in the device. Clients must specify the destination by the `zmq.IDENTITY` of the `PAIR` socket connected to the downstream end of the device.

At the Kernel level, both of these `PAIR` sockets are treated in the same way as the `REP` socket in the serial version (except using `ZMQStreams` instead of explicit sockets).

Client connections

The controller listens on an `XREP` socket for queries from clients as to queue status, and control instructions. Clients can connect to this via a `PAIR` socket or `XREQ`.

The controller publishes all registration/unregistration events via a `PUB` socket. This allows clients to stay up to date with what engines are available by subscribing to the feed with a `SUB` socket. Other processes could selectively subscribe to just registration or unregistration events.

7.12 The magic commands subsystem

Warning: These are *preliminary* notes and thoughts on the magic system, kept here for reference so we can come up with a good design now that the major core refactoring has made so much progress. Do not consider yet any part of this document final.

Two entry points:

- `m.line_eval(self,parameter_s)`: like today
- `m.block_eval(self,code_block)`: for whole-block evaluation.

This would allow us to have magics that take input, and whose single line form can even take input and call `block_eval` later (like `%cpaste` does, but with a generalized interface).

7.12.1 Constructor

Suggested syntax:

```
class MyMagic(BaseMagic):
    requires_shell = True/False
    def __init__(self, shell=None):
```

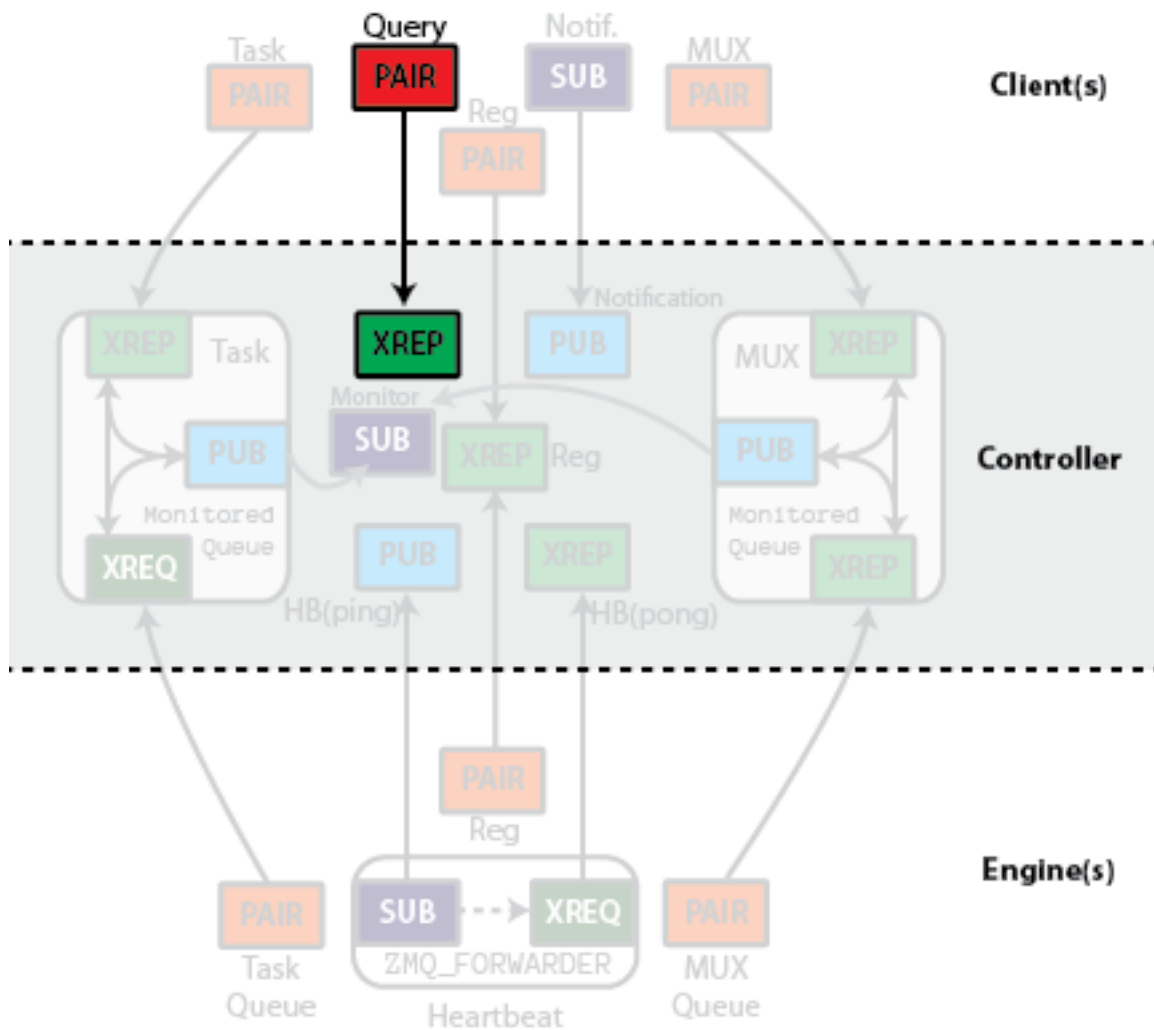



Figure 7.5: Clients connect to an XREP socket to query the controller

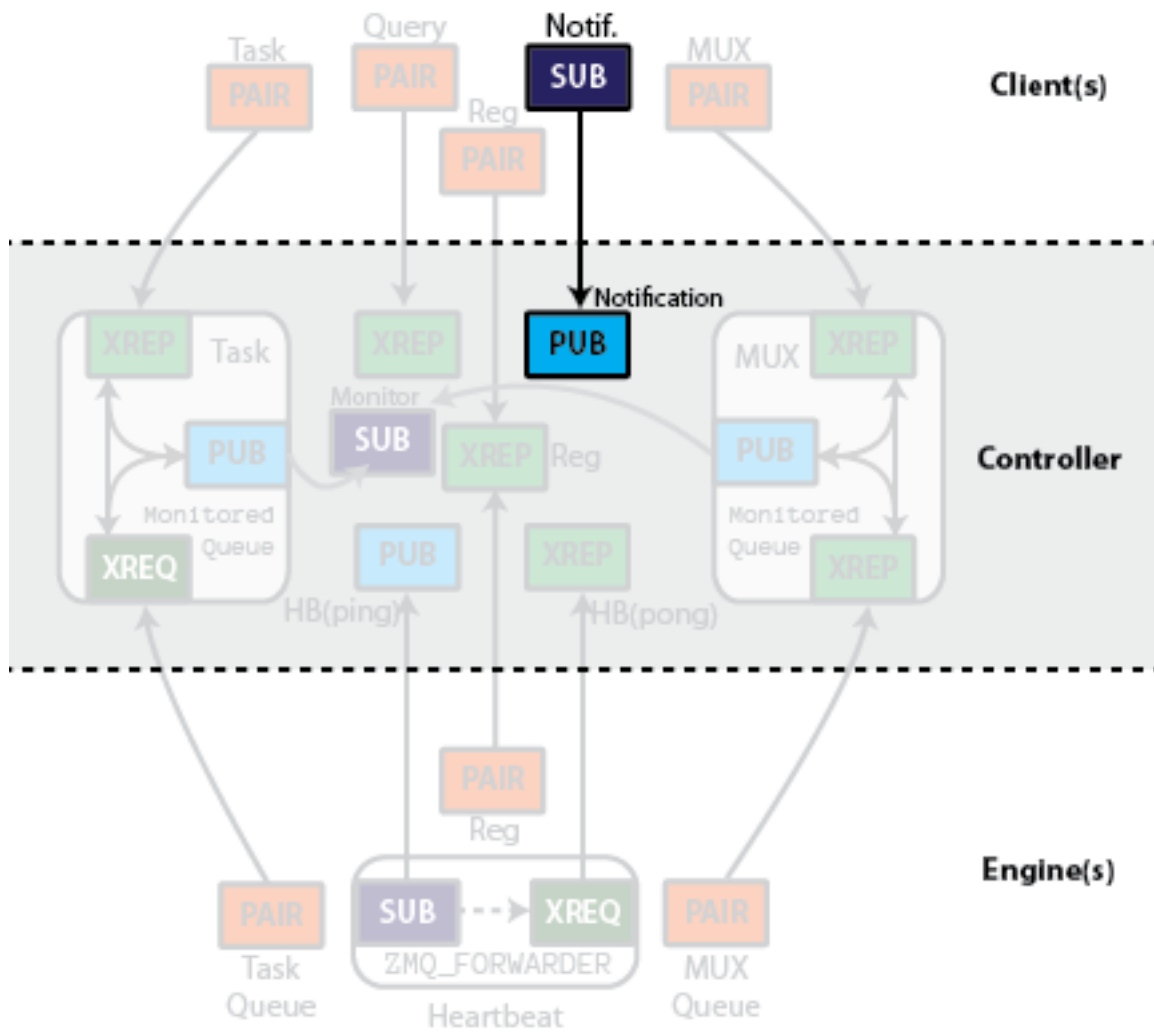


Figure 7.6: Engine registration events are published via a PUB socket.

7.12.2 Registering magics

Today, ipapi provides an *expose_magic()* function for making simple magics. We will probably extend this (in a backwards-compatible manner if possible) to allow the simplest cases to work as today, while letting users register more complex ones.

Use cases:

```
def func(arg): pass # note signature, no 'self'
ip.expose_magic('name', func)

def func_line(arg): pass
def func_block(arg): pass
ip.expose_magic('name', func_line, func_block)

class mymagic(BaseMagic):
    """Magic docstring, used in help messages.
    """
    def line_eval(self, arg): pass
    def block_eval(self, arg): pass

ip.register_magic(mymagic)
```

The BaseMagic class will offer common functionality to all, including things like options handling (via argparse).

7.12.3 Call forms: line and block

Block-oriented environments will call *line_eval()* for the first line of input (the call line starting with '%') and will then feed the rest of the block to *block_eval()* if the magic in question has a block mode.

In line environments, by default *%foo -> foo.line_eval()*, but no block call is made. Specific implementations of *line_eval* can decide to then call *block_eval* if they want to provide for whole-block input in line-oriented environments.

The api might be adapted for this decision to be made automatically by the frontend...

7.12.4 Precompiled magics for rapid loading

For IPython itself, we'll have a module of 'core' magic functions that do not require run-time registration. These will be the ones contained today in *Magic.py*, plus any others we deem worthy of being available by default. This is a trick to enable faster startup, since once we move to a model where each magic can in principle be registered at runtime, creating a lot of them can easily swamp startup time.

The trick is to make a module with a top-level class object that contains explicit references to all the 'core' magics in its dict. This way, the magic table can be quickly updated at interpreter startup with a single call, by doing something along the lines of:

```
self.magic_table.update(static_magics.__dict__)
```

The point will be to be able to bypass the explicit calling of whatever `register_magic()` API we end up making for users to declare their own magics. So ultimately one should be able to do either:

```
ip.register_magic(mymagic) # for one function
```

or:

```
ip.load_magics(static_magics) # for a bunch of them
```

I still need to clarify exactly how this should work though.

7.13 IPython.kernel.core.notification blueprint

7.13.1 Overview

The `IPython.kernel.core.notification` module will provide a simple implementation of a notification center and support for the observer pattern within the `IPython.kernel.core`. The main intended use case is to provide notification of Interpreter events to an observing frontend during the execution of a single block of code.

7.13.2 Functional Requirements

The notification center must:

- Provide synchronous notification of events to all registered observers.
- Provide typed or labeled notification types.
- Allow observers to register callbacks for individual or all notification types.
- Allow observers to register callbacks for events from individual or all notifying objects.
- Notification to the observer consists of the notification type, notifying object and user-supplied extra information [implementation: as keyword parameters to the registered callback].
- Perform as $O(1)$ in the case of no registered observers.
- Permit out-of-process or cross-network extension.

7.13.3 What's not included

As written, the `IPython.kernel.core.notification` module does not:

- Provide out-of-process or network notifications (these should be handled by a separate, Twisted aware module in `IPython.kernel`).
- Provide `zope.interface` style interfaces for the notification system (these should also be provided by the `IPython.kernel` module).

7.13.4 Use Cases

The following use cases describe the main intended uses of the notification module and illustrate the main success scenario for each use case:

Scenario 1

Dwight Schroot is writing a frontend for the IPython project. His frontend is stuck in the stone age and must communicate synchronously with an `IPython.kernel.core.Interpreter` instance. Because code is executed in blocks by the Interpreter, Dwight's UI freezes every time he executes a long block of code. To keep track of the progress of his long running block, Dwight adds the following code to his frontend's set-up code:

```
from IPython.kernel.core.notification import NotificationCenter
center = NotificationCenter.sharedNotificationCenter
center.registerObserver(self, type=IPython.kernel.core.Interpreter.STDOUT_NOTIFICATION_TYPE)
```

and elsewhere in his front end:

```
def stdout_notification(self, type, notifying_object, out_string=None):
    self.writeStdOut(out_string)
```

If everything works, the Interpreter will (according to its published API) fire a notification via the `IPython.kernel.core.notification.sharedCenter` of type `STD_OUT_NOTIFICATION_TYPE` before writing anything to stdout [it's up to the Interpreter implementation to figure out when to do this]. The notification center will then call the registered callbacks for that event type (in this case, Dwight's frontend's `stdout_notification` method). Again, according to its API, the Interpreter provides an additional keyword argument when firing the notification of `out_string`, a copy of the string it will write to stdout.

Like magic, Dwight's frontend is able to provide output, even during long-running calculations. Now if Jim could just convince Dwight to use Twisted...

Scenario 2

Boss Hog is writing a frontend for the IPython project. Because Boss Hog is stuck in the stone age, his frontend will be written in a new Fortran-like dialect of python and will run only from the command line. Because he doesn't need any fancy notification system and is used to worrying about every cycle on his rat-wheel powered mini, Boss Hog is adamant that the new notification system not produce any performance penalty. As they say in Hazard county, there's no such thing as a free lunch. If he wanted zero overhead, he should have kept using IPython 0.8. Instead, those tricky Duke boys slide in a suped-up bridge-out jumpin' awkwardly confederate-lovin' notification module that imparts only a constant (and small) performance penalty when the Interpreter (or any other object) fires an event for which there are no registered observers. Of course, the same notification-enabled Interpreter can then be used in frontends that require notifications, thus saving the IPython project from a nasty civil war.

Scenario 3

Barry is writing a frontend for the IPython project. Because Barry's front end is the *new hotness*, it uses an asynchronous event model to communicate with a Twisted `IPython.kernel.engineservice` that communicates with the IPython `IPython.kernel.core.interpreter.Interpreter`. Using the `IPython.kernel.notification` module, an asynchronous wrapper on the `IPython.kernel.core.notification` module, Barry's frontend can register for notifications from the interpreter that are delivered asynchronously. Even if Barry's frontend is running on a separate process or even host from the Interpreter, the notifications are delivered, as if by dark and twisted magic. Just like Dwight's frontend, Barry's frontend can now receive notifications of e.g. writing to stdout/stderr, opening/closing an external file, an exception in the executing code, etc.

7.14 Notes on code execution in InteractiveShell

7.14.1 Overview

This section contains information and notes about the code execution system in `InteractiveShell`. This system needs to be refactored and we are keeping notes about this process here.

7.14.2 Current design

Here is a script that shows the relationships between the various methods in `InteractiveShell` that manage code execution:

```
import networkx as nx
import matplotlib.pyplot as plt

exec_init_cmd = 'exec_init_cmd'
interact = 'interact'
runlines = 'runlines'
runsource = 'runsource'
runcode = 'runcode'
push_line = 'push_line'
mainloop = 'mainloop'
embed_mainloop = 'embed_mainloop'
ri = 'raw_input'
prefilter = 'prefilter'

g = nx.DiGraph()

g.add_node(exec_init_cmd)
g.add_node(interact)
g.add_node(runlines)
g.add_node(runsource)
g.add_node(push_line)
g.add_node(mainloop)
g.add_node(embed_mainloop)
g.add_node(ri)
```

```
g.add_node(prefilter)

g.add_edge(exec_init_cmd, push_line)
g.add_edge(exec_init_cmd, prefilter)
g.add_edge(mainloop, exec_init_cmd)
g.add_edge(mainloop, interact)
g.add_edge(embed_mainloop, interact)
g.add_edge(interact, ri)
g.add_edge(interact, push_line)
g.add_edge(push_line, runsource)
g.add_edge(runlines, push_line)
g.add_edge(runlines, prefilter)
g.add_edge(runsource, runcode)
g.add_edge(ri, prefilter)

nx.draw_spectral(g, node_size=100, alpha=0.6, node_color='r',
                font_size=10, node_shape='o')
plt.show()
```

7.15 IPython Qt interface

7.15.1 Abstract

This is about the implementation of a Qt-based Graphical User Interface (GUI) to execute Python code with an interpreter that runs in a separate process and the two systems (GUI frontend and interpreter kernel) communicating via the ZeroMQ Messaging library. The bulk of the implementation will be done without dependencies on IPython (only on Zmq). Once the key features are ready, IPython-specific features can be added using the IPython codebase.

7.15.2 Project details

For a long time there has been demand for a graphical user interface for IPython, and the project already ships Wx-based prototypes thereof. But these run all code in a single process, making them extremely brittle, as a crash of the Python interpreter kills the entire user session. Here I propose to build a Qt-based GUI that will communicate with a separate process for the code execution, so that if the interpreter kernel dies, the frontend can continue to function after restarting a new kernel (and offering the user the option to re-execute all inputs, which the frontend can know).

This GUI will allow for the easy editing of multi-line input and the convenient re-editing of previous blocks of input, which can be displayed in a 2-d workspace instead of a line-driven one like today's IPython. This makes it much easier to incrementally build and tune a code, by combining the rapid feedback cycle of IPython with the ability to edit multiline code with good graphical support.

2-process model pyzmq base

Since the necessity of a user to keep his data safe, the design is based in a 2-process model that will be achieved with a simple client/server system with `pyzmq`, so the GUI session do not crash if the the kernel

process does. This will be achieved using this test [code](#) and customizing it to the necessities of the GUI such as queue management with discrimination for different frontends connected to the same kernel and tab completion. A piece of drafted code for the kernel (server) should look like this:

```
def main():
    c = zmq.Context(1, 1)
    rep_conn = connection % port_base
    pub_conn = connection % (port_base+1)
    print >>sys.__stdout__, "Starting the kernel..."
    print >>sys.__stdout__, "On:", rep_conn, pub_conn
    session = Session(username=u'kernel')
    reply_socket = c.socket(zmq.XREP)
    reply_socket.bind(rep_conn)
    pub_socket = c.socket(zmq.PUB)
    pub_socket.bind(pub_conn)
    stdout = OutStream(session, pub_socket, u'stdout')
    stderr = OutStream(session, pub_socket, u'stderr')
    sys.stdout = stdout
    sys.stderr = stderr
    display_hook = DisplayHook(session, pub_socket)
    sys.displayhook = display_hook
    kernel = Kernel(session, reply_socket, pub_socket)
```

This kernel will use two queues (output and input), the input queue will have the id of the process(frontend) making the request, type(execute, complete, help, etc) and id of the request itself and the string of code to be executed, the output queue will have basically the same information just that the string is the to be displayed. This model is because the kernel needs to maintain control of timeouts when multiple requests are sent and keep them indexed.

Qt based GUI

Design of the interface is going to be based in cells of code executed on the previous defined kernel. It will also have GUI facilities such toolboxes, tooltips to autocomplete code and function summary, highlighting and autoindentation. It will have the cell kind of multiline edition mode so each block of code can be edited and executed independently, this can be achieved queuing QTextEdit objects (the cell) giving them format so we can discriminate outputs from inputs. One of the main characteristics will be the debug support that will show the requested outputs as the debugger (that will be on a popup widget) “walks” through the code, this design is to be reviewed with the mentor. [This](#) is a tentative view of the main window.

The GUI will check continuously the output queue from the kernel for new information to handle. This information have to be handled with care since any output will come at anytime and possibly in a different order than requested or maybe not appear at all, this could be possible due to a variety of reasons(for example tab completion request while the kernel is busy processing another frontend’s request). This is, if the kernel is busy it won’t be possible to fulfill the request for a while so the GUI will be prepared to abandon waiting for the reply if the user moves on or a certain timeout expires.

7.15.3 POSSIBLE FUTURE DIRECTIONS

The near future will bring the feature of saving and loading sessions, also importing and exporting to different formats like rst, html, pdf and python/ipython code, a discussion about this is taking place in the

ipython-dev mailing list. Also the interaction with a remote kernel and distributed computation which is an IPython's project already in development.

The idea of a mathematica-like help widget (i.e. there will be parts of it that will execute as a native session of IPythonQt) is still to be discussed in the development mailing list but it's definitively a great idea.

7.16 Porting IPython to a two process model using zeromq

7.16.1 Abstract

IPython's execution in a command-line environment will be ported to a two process model using the zeromq library for inter-process communication. this will:

- prevent an interpreter crash from destroying the user session,
- allow multiple clients to interact simultaneously with a single interpreter
- allow IPython to reuse code for local execution and distributed computing (dc)
- give us a path for python3 support, since zeromq supports python3 while twisted (what we use today for dc) does not.

7.16.2 Project description

Currently IPython provides a command-line client that executes all code in a single process, and a set of tools for distributed and parallel computing that execute code in multiple processes (possibly but not necessarily on different hosts), using the twisted asynchronous framework for communication between nodes. for a number of reasons, it is desirable to unify the architecture of the local execution with that of distributed computing, since ultimately many of the underlying abstractions are similar and should be reused. in particular, we would like to:

- have even for a single user a 2-process model, so that the environment where code is being input runs in a different process from that which executes the code. this would prevent a crash of the python interpreter executing code (because of a segmentation fault in a compiled extension or an improper access to a c library via ctypes, for example) from destroying the user session.
- have the same kernel used for executing code locally be available over the network for distributed computing. currently the twisted-using IPython engines for distributed computing do not share any code with the command-line client, which means that many of the additional features of IPython (tab completion, object introspection, magic functions, etc) are not available while using the distributed computing system. once the regular command-line environment is ported to allowing such a 2-process model, this newly decoupled kernel could form the core of a distributed computing IPython engine and all capabilities would be available throughout the system.
- have a route to python3 support. twisted is a large and complex library that does currently not support python3, and as indicated by the twisted developers it may take a while before it is ported (<http://stackoverflow.com/questions/172306/how-are-you-planning-on-handling-the-migration-to-python-3>). for IPython, this means that while we could port the command-line environment, a large swath of IPython would be left 2.x-only, a highly undesirable situation. for this reason,

the search for an alternative to twisted has been active for a while, and recently we've identified the zeromq (<http://www.zeromq.org>, zmq for short) library as a viable candidate. zmq is a fast, simple messaging library written in c++, for which one of the IPython developers has written python bindings using cython (<http://www.zeromq.org/bindings:python>). since cython already knows how to generate python3-compliant bindings with a simple command-line switch, zmq can be used with python3 when needed.

As part of the zmq python bindings, the IPython developers have already developed a simple prototype of such a two-process kernel/frontend system (details below). I propose to start from this example and port today's IPython code to operate in a similar manner. IPython's command-line program (the main 'ipython' script) executes both user interaction and the user's code in the same process. This project will thus require breaking up IPython into the parts that correspond to the kernel and the parts that are meant to interact with the user, and making these two components communicate over the network using zmq instead of accessing local attributes and methods of a single global object.

Once this port is complete, the resulting tools will be the foundation (though as part of this proposal i do not expect to undertake either of these tasks) to allow the distributed computing parts of IPython to use the same code as the command-line client, and for the whole system to be ported to python3. so while i do not intend to tackle here the removal of twisted and the unification of the local and distributed parts of IPython, my proposal is a necessary step before those are possible.

7.16.3 Project details

As part of the zeromq bindings, the IPython developers have already developed a simple prototype example that provides a python execution kernel (with none of IPython's code or features, just plain code execution) that listens on zmq sockets, and a frontend based on the interactiveconsole class of the code.py module from the python standard library. this example is capable of executing code, propagating errors, performing tab-completion over the network and having multiple frontends connect and disconnect simultaneously to a single kernel, with all inputs and outputs being made available to all connected clients (thanks to zmq's pub sockets that provide multicasting capabilities for the kernel and to which the frontends subscribe via a sub socket).

We have all example code in:

- <http://github.com/ellisonbg/pyzmq/blob/master/examples/kernel/kernel.py>
- <http://github.com/ellisonbg/pyzmq/blob/master/examples/kernel/completer.py>
- <http://github.com/fperez/pyzmq/blob/master/examples/kernel/frontend.py>

all of this code already works, and can be seen in this example directory from the zmq python bindings:

- <http://github.com/ellisonbg/pyzmq/blob/master/examples/kernel>

Based on this work, i expect to write a stable system for IPython kernel with IPython standards, error control, crash recovery system and general configuration options, also standardize defaults ports or auth system for remote connection etc.

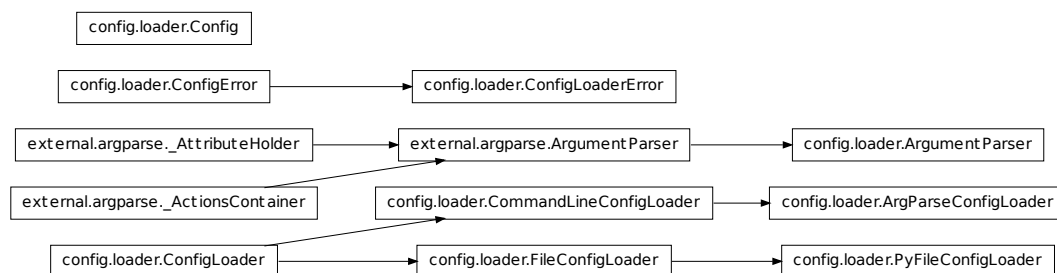
The crash recovery system, is a IPython kernel module for when it fails unexpectedly, you can retrieve the information from the section, this will be based on a log and a lock file to indicate when the kernel was not closed in a proper way.

THE IPYTHON API

8.1 config.loader

8.1.1 Module: config.loader

Inheritance diagram for `IPython.config.loader`:



A simple configuration system.

Authors

- Brian Granger
- Fernando Perez

8.1.2 Classes

`ArgParseConfigLoader`

```
class IPython.config.loader.ArgParseConfigLoader (argv=None, *parser_args,
                                                  **parser_kw)
    Bases: IPython.config.loader.CommandLineConfigLoader
```

```
__init__ (argv=None, *parser_args, **parser_kw)
```

Create a config loader for use with argparse.

Parameters **argv** : optional, list

If given, used to read command-line arguments from, otherwise sys.argv[1:] is used.

parser_args : tuple

A tuple of positional arguments that will be passed to the constructor of argparse.ArgumentParser.

parser_kw : dict

A tuple of keyword arguments that will be passed to the constructor of argparse.ArgumentParser.

```
clear ()
```

```
get_extra_args ()
```

```
load_config (args=None)
```

Parse command line arguments and return as a Struct.

Parameters **args** : optional, list

If given, a list with the structure of sys.argv[1:] to parse arguments from. If not given, the instance's self.argv attribute (given at construction time) is used.

ArgumentParser

```
class IPython.config.loader.ArgumentParser (prog=None,      usage=None,      de-
                                             description=None,      epilog=None,
                                             version=None,      parents=[
                                             ],      formatter_class=<class
                                             'IPython.external.argparse.HelpFormatter'>,
                                             prefix_chars='-',      from-
                                             file_prefix_chars=None,      ar-
                                             gument_default=None,      con-
                                             flict_handler='error', add_help=True)
```

Bases: IPython.external.argparse.ArgumentParser

Simple argparse subclass that prints help to stdout by default.

```
__init__ (prog=None, usage=None, description=None, epilog=None, version=None, par-
           ents=[], formatter_class=<class 'IPython.external.argparse.HelpFormatter'>,
           prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, con-
           flict_handler='error', add_help=True)
```

```
add_argument (dest, ..., name=value, ...) add_argument(option_string, option_string, ...,
                                                         name=value, ...)
```

```
add_argument_group (*args, **kwargs)
```

```

add_mutually_exclusive_group (**kwargs)
add_subparsers (**kwargs)
error (message: string)
    Prints a usage message incorporating the message to stderr and exits.

    If you override this in a subclass, it should not return – it should either exit or raise an exception.
exit (status=0, message=None)
format_help ()
format_usage ()
format_version ()
get_default (dest)
parse_args (args=None, namespace=None)
parse_known_args (args=None, namespace=None)
print_help (file=None)
print_usage (file=None)
print_version (file=None)
register (registry_name, value, object)
set_defaults (**kwargs)

```

CommandLineConfigLoader

class IPython.config.loader.**CommandLineConfigLoader**

Bases: IPython.config.loader.ConfigLoader

A config loader for command line arguments.

As we add more command line based loaders, the common logic should go here.

```

__init__ ()
    A base class for config loaders.

```

Examples

```

>>> cl = ConfigLoader()
>>> config = cl.load_config()
>>> config
{}

```

```

clear ()

```

load_config()

Load a config from somewhere, return a `Config` instance.

Usually, this will cause `self.config` to be set and then returned. However, in most cases, `ConfigLoader.clear()` should be called to erase any previous state.

Config

class `IPython.config.loader.Config(*args, **kws)`

Bases: `dict`

An attribute based dict that can do smart merges.

__init__ (*args, **kws)

clear

D.clear() -> None. Remove all items from D.

copy ()

static fromkeys ()

dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.

get

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

has_key (key)

items

D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems

D.iteritems() -> an iterator over the (key, value) items of D

iterkeys

D.iterkeys() -> an iterator over the keys of D

itervalues

D.itervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise `KeyError` is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if D is empty.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does: for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

ConfigError

class IPython.config.loader.**ConfigError**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

ConfigLoader

class IPython.config.loader.**ConfigLoader**

Bases: object

A object for loading configurations from just about anywhere.

The resulting configuration is packaged as a Struct.

Notes

A **ConfigLoader** does one thing: load a config from a source (file, command line arguments) and returns the data as a Struct. There are lots of things that **ConfigLoader** does not do. It does not implement complex logic for finding config files. It does not handle default values or merge multiple configs. These things need to be handled elsewhere.

__init__()

A base class for config loaders.

Examples

```
>>> cl = ConfigLoader()
>>> config = cl.load_config()
>>> config
{'}
```

clear()

load_config()

Load a config from somewhere, return a `Config` instance.

Usually, this will cause `self.config` to be set and then returned. However, in most cases, `ConfigLoader.clear()` should be called to erase any previous state.

ConfigLoaderError

class IPython.config.loader.**ConfigLoaderError**

Bases: IPython.config.loader.ConfigError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

FileConfigLoader

class IPython.config.loader.**FileConfigLoader**

Bases: IPython.config.loader.ConfigLoader

A base class for file based configurations.

As we add more file based config loaders, the common logic should go here.

__init__()

A base class for config loaders.

Examples

```
>>> cl = ConfigLoader()
>>> config = cl.load_config()
>>> config
{}
```

clear()

load_config()

Load a config from somewhere, return a `Config` instance.

Usually, this will cause `self.config` to be set and then returned. However, in most cases, `ConfigLoader.clear()` should be called to erase any previous state.

PyFileConfigLoader

class IPython.config.loader.**PyFileConfigLoader** (*filename, path=None*)

Bases: IPython.config.loader.FileConfigLoader

A config loader for pure python files.

This calls `execfile` on a plain python file and looks for attributes that are all caps. These attribute are added to the config Struct.

```
__init__(filename, path=None)
    Build a config loader for a filename and path.
```

Parameters **filename** : str

The file name of the config file.

path : str, list, tuple

The path to search for the config file on, or a sequence of paths to try in order.

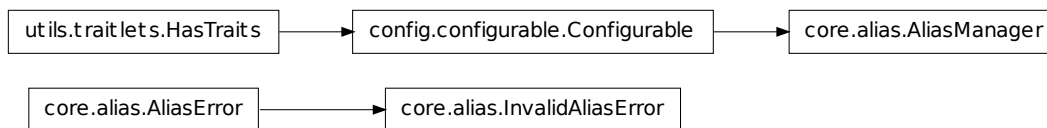
```
clear()
```

```
load_config()
    Load the config from a file and return it as a Struct.
```

8.2 core.alias

8.2.1 Module: `core.alias`

Inheritance diagram for `IPython.core.alias`:



System command aliases.

Authors:

- Fernando Perez
- Brian Granger

8.2.2 Classes

`AliasError`

```
class IPython.core.alias.AliasError
    Bases: exceptions.Exception
```

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

AliasManager

class IPython.core.alias.**AliasManager** (*shell=None, config=None*)
Bases: IPython.config.configurable.Configurable

__init__ (*shell=None, config=None*)

aliases

call_alias (*alias, rest=''*)
Call an alias given its name and the rest of the line.

clear_aliases ()

config
A trait whose value must be an instance of a specified class.
The value can also be an instance of a subclass of the specified class.

default_aliases
An instance of a Python list.

define_alias (*name, cmd*)
Define a new alias after validating it.
This will raise an `AliasError` if there are validation problems.

exclude_aliases ()

expand_alias (*line*)
Expand an alias in the command line
Returns the provided command line, possibly with the first word (command) translated according to alias expansion rules.

[ipython]|16> **_ip.expand_aliases**("np myfile.txt") <16> 'q:/opt/np/notepad++.exe my-file.txt'

expand_aliases (*fn, rest*)
Expand multiple levels of aliases:
if:
alias foo bar /tmp alias baz foo
then:
baz huhhahhei -> bar /tmp huhhahhei

init_aliases ()

on_trait_change (*handler*, *name=None*, *remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

soft_define_alias (*name*, *cmd*)

Define an alias, but don’t raise on an AliasError.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

transform_alias (*alias*, *rest=’**’*)

Transform alias to system command string.

undefine_alias (*name*)

user_aliases

An instance of a Python list.

validate_alias (*name*, *cmd*)

Validate an alias and return the its number of arguments.

InvalidAliasError

```
class IPython.core.alias.InvalidAliasError
    Bases: IPython.core.alias.AliasError
    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature
    args
    message
```

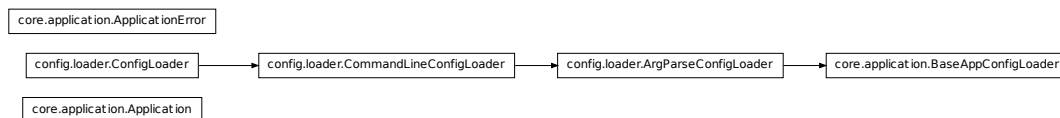
8.2.3 Function

```
IPython.core.alias.default_aliases()
    Return list of shell aliases to auto-define.
```

8.3 core.application

8.3.1 Module: core.application

Inheritance diagram for `IPython.core.application`:



An application for IPython.

All top-level applications should use the classes in this module for handling configuration and creating componentnets.

The job of an `Application` is to create the master configuration object and then create the configurable objects, passing the config to them.

Authors:

- Brian Granger
- Fernando Perez

Notes

8.3.2 Classes

Application

class IPython.core.application.**Application** (*argv=None*)

Bases: object

Load a config, construct configurables and set them running.

The configuration of an application can be done via three different Config objects, which are loaded and ultimately merged into a single one used from that point on by the app. These are:

- 1.default_config: internal defaults, implemented in code.
- 2.file_config: read from the filesystem.
- 3.command_line_config: read from the system's command line flags.

During initialization, 3 is actually read before 2, since at the command-line one may override the location of the file to be read. But the above is the order in which the merge is made.

__init__ (*argv=None*)

argv

A reference to the argv to be used (typically ends up being sys.argv[1:])

attempt (*func*)

command_line_config

Read from the system's command line flags.

command_line_loader

The command line config loader. Subclass of ArgParseConfigLoader.

config_file_name

The name of the config file to load, determined at runtime

construct ()

Construct the main objects that make up this app.

crash_handler_class

The class to use as the crash handler.

create_command_line_config ()

Create and return a command line config loader.

create_crash_handler ()

Create a crash handler, typically setting sys.excepthook to it.

create_default_config ()

Create defaults that can't be set elsewhere.

For the most part, we try to set default in the class attributes of Configurables. But, defaults the top-level Application (which is not a HasTraits or Configurables) are not set in this way. Instead

we set them here. The Global section is for variables like this that don't belong to a particular configurable.

default_config

Internal defaults, implemented in code.

default_config_file_name

The name of the default config file. Track separately from the actual name because some logic happens only if we aren't using the default.

exit (*exit_status=0*)**extra_args**

extra arguments computed by the command-line loader

file_config

Read from the filesystem.

find_config_file_name ()

Find the config file name for this application.

This must set `self.config_file_name` to the filename of the config file to use (just the filename). The search paths for the config file are set in `find_config_file_paths()` and then passed to the config file loader where they are resolved to an absolute path.

If a profile has been set at the command line, this will resolve it.

find_config_file_paths ()

Set the search paths for resolving the config file.

This must set `self.config_file_paths` to a sequence of search paths to pass to the config file loader.

find_ipython_dir ()

Set the IPython directory.

This sets `self.ipython_dir`, but the actual value that is passed to the application is kept in either `self.default_config` or `self.command_line_config`. This also adds `self.ipython_dir` to `sys.path` so config files there can be referenced by other config files.

find_resources ()

Find other resources that need to be in place.

Things like cluster directories need to be in place to find the config file. These happen right after the IPython directory has been set.

init_logger ()**initialize** ()

Initialize the application.

Loads all configuration information and sets all application state, but does not start any relevant processing (typically some kind of event loop).

Once this method has been called, the application is flagged as initialized and the method becomes a no-op.

ipython_dir

User's ipython directory, typically ~/.ipython/

load_command_line_config()

Load the command line config.

load_file_config()

Load the config file.

This tries to load the config file from disk. If successful, the `CONFIG_FILE` config variable is set to the resolved config file location. If not successful, an empty config is used.

log_command_line_config()

log_default_config()

log_file_config()

log_level

log_master_config()

master_config

The final config that will be passed to the main object.

merge_configs()

Merge the default, command line and file config objects.

post_construct()

Do actions after construct, but before starting the app.

post_load_command_line_config()

Do actions just after loading the command line config.

post_load_file_config()

Do actions after the config file is loaded.

pre_construct()

Do actions after the config has been built, but before construct.

pre_load_command_line_config()

Do actions just before loading the command line config.

pre_load_file_config()

Do actions before the config file is loaded.

profile_name

Set by `-profile` option

set_command_line_config_log_level()

set_default_config_log_level()

set_file_config_log_level()

start()

Start the application.

start_app()

Actually start the app.

usage

Usage message printed by argparse. If None, auto-generate

ApplicationError

class IPython.core.application.**ApplicationError**

Bases: `exceptions.Exception`

__init__()

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

args

message

BaseAppConfigLoader

class IPython.core.application.**BaseAppConfigLoader** (*argv=None*, **parser_args*,
***parser_kw*)

Bases: `IPython.config.loader.ArgParseConfigLoader`

Default command line options for IPython based applications.

__init__ (*argv=None*, **parser_args*, ***parser_kw*)

Create a config loader for use with argparse.

Parameters **argv** : optional, list

If given, used to read command-line arguments from, otherwise `sys.argv[1:]` is used.

parser_args : tuple

A tuple of positional arguments that will be passed to the constructor of `argparse.ArgumentParser`.

parser_kw : dict

A tuple of keyword arguments that will be passed to the constructor of `argparse.ArgumentParser`.

clear()

get_extra_args()

load_config (*args=None*)

Parse command line arguments and return as a Struct.


Parameters **args** : optional, list

If given, a list with the structure of `sys.argv[1:]` to parse arguments from. If not given, the instance's `self.argv` attribute (given at construction time) is used.

8.4 core.autocall

8.4.1 Module: `core.autocall`

Inheritance diagram for `IPython.core.autocall`:



```
graph TD; A[core.autocall.IPyAutocall];
```

core.autocall.IPyAutocall

Autocall capabilities for `IPython.core`.

Authors:

- Brian Granger
- Fernando Perez

Notes

8.4.2 `IPyAutocall`

class `IPython.core.autocall.IPyAutocall`

Bases: `object`

Instances of this class are always autocalled

This happens regardless of ‘autocall’ variable state. Use this to develop macro-like mechanisms.

__init__ ()

x.**__init__**(...) initializes x; see x.**__class__**.**__doc__** for signature

set_ip (ip)

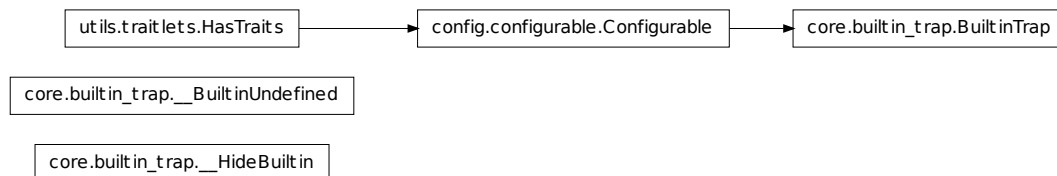
Will be used to set `_ip` point to current ipython instance b/f call

Override this method if you don’t want this to happen.

8.5 core.builtin_trap

8.5.1 Module: core.builtin_trap

Inheritance diagram for `IPython.core.builtin_trap`:



A context manager for managing things injected into `__builtin__`.

Authors:

- Brian Granger
- Fernando Perez

8.5.2 BuiltinTrap

class `IPython.core.builtin_trap.BuiltinTrap` (*shell=None*)

Bases: `IPython.config.configurable.Configurable`

__init__ (*shell=None*)

activate ()

Store ipython references in the `__builtin__` namespace.

add_builtin (*key, value*)

Add a builtin and save the original.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deactivate ()

Remove any builtins which might have been added by `add_builtins`, or restore overwritten ones to their previous values.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

remove_builtin (*key*)

Remove an added builtin and re-set the original.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

8.6 core.compilerop

8.6.1 Module: `core.compilerop`

Inheritance diagram for `IPython.core.compilerop`:

core.compilerop.CachingCompiler

Compiler tools with improved interactive support.

Provides compilation machinery similar to codeop, but with caching support so we can provide interactive tracebacks.

Authors

- Robert Kern
- Fernando Perez

8.6.2 CachingCompiler

class IPython.core.compilerop.CachingCompiler

Bases: object

A compiler that caches code compiled from interactive statements.

__init__()

check_cache(*args)

Call linecache.checkcache() safely protecting our cached values.

compiler_flags

Flags currently active in the compilation process.

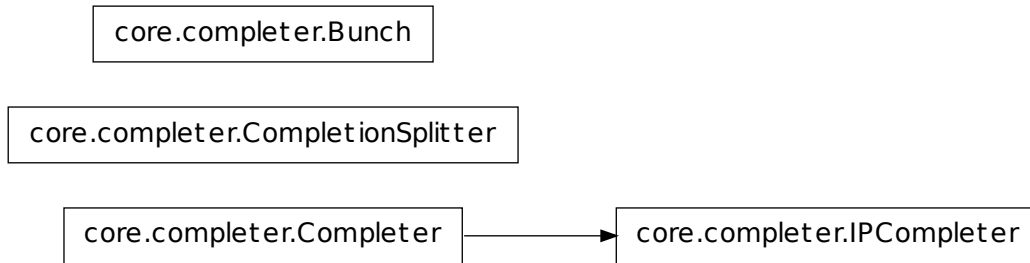
IPython.core.compilerop.**code_name**(code, number=0)

Compute a (probably) unique name for code for caching.

8.7 core.completer

8.7.1 Module: core.completer

Inheritance diagram for IPython.core.completer:



Word completion for IPython.

This module is a fork of the `rlcompleter` module in the Python standard library. The original enhancements made to `rlcompleter` have been sent upstream and were accepted as of Python 2.3, but we need a lot more functionality specific to IPython, so this module will continue to live as an IPython-specific utility.

Original `rlcompleter` documentation:

This requires the latest extension to the `readline` module (the completes keywords, built-ins and globals in `__main__`; when completing `NAME.NAME...`, it evaluates (!) the expression up to the last dot and completes its attributes.

It's very cool to do "import string" type "string.", hit the completion key (twice), and see the list of names defined by the string module!

Tip: to use the tab key as the completion key, call

```
readline.parse_and_bind("tab: complete")
```

Notes:

- Exceptions raised by the completer function are *ignored* (and generally cause the completion to fail). This is a feature – since `readline` sets the tty device in raw (or cbreak) mode, printing a traceback wouldn't work well without some complicated hoopla to save, reset and restore the tty state.

- The evaluation of the `NAME.NAME...` form may cause arbitrary application defined code to be executed if an object with a `__getattr__` hook is found. Since it is the responsibility of the application (or the user) to enable this feature, I consider this an acceptable risk. More complicated expressions (e.g. function calls or indexing operations) are *not* evaluated.

- GNU `readline` is also used by the built-in functions `input()` and `raw_input()`, and thus these also benefit/suffer from the completer features. Clearly an interactive application can benefit by specifying its own completer function and using `raw_input()` for all its input.

- When the original `stdin` is not a tty device, GNU `readline` is never used, and this module (and the `readline` module) are silently inactive.

8.7.2 Classes

Bunch

class IPython.core.completer.**Bunch**

Bases: object

__init__()

x.**__init__**(...) initializes x; see x.**__class__**.**__doc__** for signature

Completer

class IPython.core.completer.**Completer**(*namespace=None*,

global_namespace=None)

Bases: object

__init__(*namespace=None*, *global_namespace=None*)

Create a new completer for the command line.

Completer([*namespace*, *global_namespace*]) -> completer instance.

If unspecified, the default namespace where completions are performed is **__main__** (technically, **__main__**.**__dict__**). Namespaces should be given as dictionaries.

An optional second namespace can be given. This allows the completer to handle cases where both the local and global scopes need to be distinguished.

Completer instances should be used as the completion mechanism of readline via the **set_completer**() call:

```
readline.set_completer(Completer(my_namespace).complete)
```

attr_matches(*text*)

Compute matches when text contains a dot.

Assuming the text is of the form NAME.NAME...[NAME], and is evaluable in **self.namespace** or **self.global_namespace**, it will be evaluated and its attributes (as revealed by **dir()**) are used as possible completions. (For class instances, class members are also considered.)

WARNING: this can still invoke arbitrary C code, if an object with a **__getattr__** hook is evaluated.

complete(*text*, *state*)

Return the next possible completion for 'text'.

This is called successively with **state** == 0, 1, 2, ... until it returns None. The completion should begin with 'text'.

global_matches(*text*)

Compute matches when text is a simple name.

Return a list of all keywords, built-in functions and names currently defined in **self.namespace** or **self.global_namespace** that match.

CompletionSplitter

class IPython.core.completer.**CompletionSplitter** (*delims=None*)

Bases: object

An object to split an input line in a manner similar to readline.

By having our own implementation, we can expose readline-like completion in a uniform manner to all frontends. This object only needs to be given the line of text to be split and the cursor position on said line, and it returns the ‘word’ to be completed on at the cursor after splitting the entire line.

What characters are used as splitting delimiters can be controlled by setting the *delims* attribute (this is a property that internally automatically builds the necessary

__init__ (*delims=None*)

get_delims ()

Return the string of delimiter characters.

set_delims (*delims*)

Set the delimiters for line splitting.

split_line (*line, cursor_pos=None*)

Split a line of text with a cursor at the given position.

IPCompleter

class IPython.core.completer.**IPCompleter** (*shell,* *namespace=None,*
global_namespace=None,
omit__names=True, *alias_table=None,*
use_readline=True)

Bases: IPython.core.completer.Completer

Extension of the completer class with IPython-specific features

__init__ (*shell,* *namespace=None,* *global_namespace=None,* *omit__names=True,*
alias_table=None, use_readline=True)

IPCompleter() -> completer

Return a completer object suitable for use by the readline library via readline.set_completer().

Inputs:

- shell: a pointer to the ipython shell itself. This is needed

because this completer knows about magic functions, and those can only be accessed via the ipython instance.

- namespace: an optional dict where completions are performed.
- global_namespace: secondary optional dict for completions, to

handle cases (such as IPython embedded inside functions) where both Python scopes are visible.

- The optional omit__names parameter sets the completer to omit the

‘magic’ names (`__magicname__`) for python objects unless the text to be completed explicitly starts with one or more underscores.

- If `alias_table` is supplied, it should be a dictionary of aliases to complete.

use_readline [bool, optional] If true, use the readline library. This completer can still function without readline, though in that case callers must provide some extra information on each call about the current line.

alias_matches (*text*)

Match internal system aliases

all_completions (*text*)

Return all possible completions for the benefit of emacs.

attr_matches (*text*)

Compute matches when text contains a dot.

Assuming the text is of the form `NAME.NAME...[NAME]`, and is evaluatable in `self.namespace` or `self.global_namespace`, it will be evaluated and its attributes (as revealed by `dir()`) are used as possible completions. (For class instances, class members are also considered.)

WARNING: this can still invoke arbitrary C code, if an object with a `__getattr__` hook is evaluated.

complete (*text=None, line_buffer=None, cursor_pos=None*)

Find completions for the given text and line context.

This is called successively with `state == 0, 1, 2, ...` until it returns `None`. The completion should begin with ‘text’.

Note that both the `text` and the `line_buffer` are optional, but at least one of them must be given.

Parameters **text** : string, optional

Text to perform the completion on. If not given, the line buffer is split using the instance’s `CompletionSplitter` object.

line_buffer [string, optional] If not given, the completer attempts to obtain the current line buffer via `readline`. This keyword allows clients which are requesting for text completions in non-readline contexts to inform the completer of the entire text.

cursor_pos [int, optional] Index of the cursor in the full line buffer. Should be provided by remote frontends where kernel has no access to frontend state.

Returns **text** : str

Text that was actually used in the completion.

matches : list

A list of completion matches.

dispatch_custom_completer (*text*)

file_matches (*text*)

Match filenames, expanding ~USER type strings.

Most of the seemingly convoluted logic in this completer is an attempt to handle filenames with spaces in them. And yet it's not quite perfect, because Python's readline doesn't expose all of the GNU readline details needed for this to be done correctly.

For a filename with a space in it, the printed completions will be only the parts after what's already been typed (instead of the full completions, as is normally done). I don't think with the current (as of Python 2.3) Python readline it's possible to do better.

global_matches (*text*)

Compute matches when text is a simple name.

Return a list of all keywords, built-in functions and names currently defined in self.namespace or self.global_namespace that match.

magic_matches (*text*)

Match magics

python_func_kw_matches (*text*)

Match named parameters (kwargs) of the last open function

python_matches (*text*)

Match attributes or global python names

rlcomplete (*text*, *state*)

Return the state-th possible completion for 'text'.

This is called successively with state == 0, 1, 2, ... until it returns None. The completion should begin with 'text'.

Parameters **text** : string

Text to perform the completion on.

state [int] Counter used by readline.

8.7.3 Functions

IPython.core.completer.**compress_user** (*path*, *tilde_expand*, *tilde_val*)

Does the opposite of expand_user, with its outputs.

IPython.core.completer.**expand_user** (*path*)

Expand '~'-style usernames in strings.

This is similar to `os.path.expanduser()`, but it computes and returns extra information that will be useful if the input was being used in computing completions, and you wish to return the completions with the original '~' instead of its expanded value.

Parameters **path** : str

String to be expanded. If no ~ is present, the output is the same as the input.

Returns `newpath` : str

Result of ~ expansion in the input path.

tilde_expand : bool

Whether any expansion was performed or not.

tilde_val : str

The value that ~ was replaced with.

`IPython.core.completer.has_open_quotes(s)`

Return whether a string has open quotes.

This simply counts whether the number of quote characters of either type in the string is odd.

Returns If there is an open quote, the quote character is returned. Else, return :

False. :

`IPython.core.completer.mark_dirs(matches)`

Mark directories in input list by appending '/' to their names.

`IPython.core.completer.protect_filename(s)`

Escape a string to protect certain characters.

`IPython.core.completer.single_dir_expand(matches)`

Recursively expand match lists containing a single dir.

8.8 core.completerlib

8.8.1 Module: core.completerlib

Implementations for various useful completers.

These are all loaded by default by IPython.

8.8.2 Functions

`IPython.core.completerlib.cd_completer(self, event)`

Completer function for cd, which only returns directories.

`IPython.core.completerlib.get_root_modules()`

Returns a list containing the names of all the modules available in the folders of the pythonpath.

`IPython.core.completerlib.is_importable(module, attr, only_modules)`

`IPython.core.completerlib.magic_run_completer(self, event)`

Complete files that end in .py or .ipy for the %run command.

`IPython.core.completerlib.module_completer(self, event)`

Give completions after user has typed 'import ...' or 'from ...'

`IPython.core.completerlib.module_completion` (*line*)

Returns a list containing the completion possibilities for an import line.

The line looks like this : 'import xml.d' 'from xml.dom import'

`IPython.core.completerlib.module_list` (*path*)

Return the list containing the names of the modules available in the given folder.

`IPython.core.completerlib.quick_completer` (*cmd*, *completions*)

Easily create a trivial completer for a command.

Takes either a list of completions, or all completions in string (that will be split on whitespace).

Example:

```
[d:\ipython]|1> import ipy_completers
[d:\ipython]|2> ipy_completers.quick_completer('foo', ['bar','baz'])
[d:\ipython]|3> foo b<TAB>
bar baz
[d:\ipython]|3> foo ba
```

`IPython.core.completerlib.shlex_split` (*x*)

Helper function to split lines into segments.

`IPython.core.completerlib.try_import` (*mod*, *only_modules=False*)

8.9 core.crashhandler

8.9.1 Module: `core.crashhandler`

Inheritance diagram for `IPython.core.crashhandler`:



```
graph TD
    A[core.crashhandler.CrashHandler]
```

`sys.excepthook` for IPython itself, leaves a detailed report on disk.

Authors:

- Fernando Perez
- Brian E. Granger

8.9.2 CrashHandler

```
class IPython.core.crashhandler.CrashHandler(app, contact_name=None,
                                             contact_email=None,
                                             bug_tracker=None,
                                             show_crash_traceback=True,
                                             call_pdb=False)
```

Bases: object

Customizable crash handlers for IPython applications.

Instances of this class provide a `__call__()` method which can be used as a `sys.excepthook`. The `__call__()` signature is:

```
def __call__(self, etype, evalue, etb)
```

```
__init__(app, contact_name=None, contact_email=None, bug_tracker=None,
          show_crash_traceback=True, call_pdb=False)
```

Create a new crash handler

Parameters **app** : Application

A running `Application` instance, which will be queried at crash time for internal information.

contact_name : str

A string with the name of the person to contact.

contact_email : str

A string with the email address of the contact.

bug_tracker : str

A string with the URL for your project's bug tracker.

show_crash_traceback : bool

If false, don't print the crash traceback on stderr, only generate the on-disk report

Non-argument instance attributes: :

These instances contain some non-argument attributes which allow for :

further customization of the crash handler's behavior. Please see the :

source for further details. :

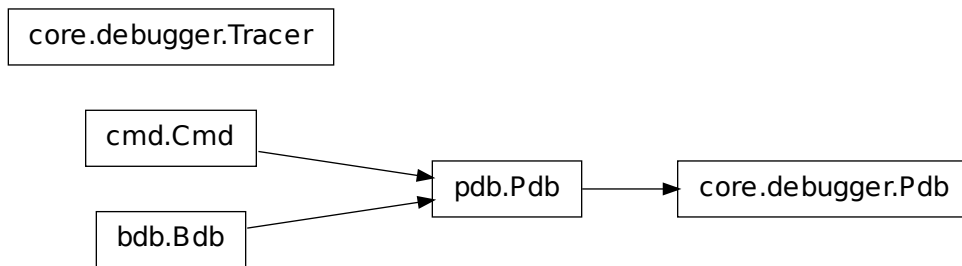
make_report (traceback)

Return a string containing a crash report.

8.10 core.debugger

8.10.1 Module: `core.debugger`

Inheritance diagram for `IPython.core.debugger`:



Pdb debugger class.

Modified from the standard `pdb.Pdb` class to avoid including `readline`, so that the command line completion of other programs which include this isn't damaged.

In the future, this class will be expanded with improvements over the standard `pdb`.

The code in this file is mainly lifted out of `cmd.py` in Python 2.2, with minor changes. Licensing should therefore be under the standard Python terms. For details on the PSF (Python Software Foundation) standard license, see:

<http://www.python.org/2.2.3/license.html>

8.10.2 Classes

Pdb

```
class IPython.core.debugger.Pdb(color_scheme='NoColor', completekey=None,
                               stdin=None, stdout=None)
```

Bases: `pdb.Pdb`

Modified Pdb class, does not load `readline`.

```
__init__(color_scheme='NoColor', completekey=None, stdin=None, stdout=None)
```

```
bp_commands(frame)
```

Call every command that was set for the current active breakpoint (if there is one) Returns True if the normal interaction function must be called, False otherwise

```
break_anywhere(frame)
```

```
break_here(frame)
```

canonic (*filename*)

checkline (*filename*, *lineno*)

Check whether specified line seems to be executable.

Return *lineno* if it is, 0 if not (e.g. a docstring, comment, blank line or EOF). Warning: testing is not comprehensive.

clear_all_breaks ()

clear_all_file_breaks (*filename*)

clear_bpbynumber (*arg*)

clear_break (*filename*, *lineno*)

cmdloop (*intro=None*)

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

columnize (*list*, *displaywidth=80*)

Display a list of strings as a compact set of columns.

Each column is only as wide as necessary. Columns are separated by two spaces (one was not legible enough).

complete (*text*, *state*)

Return the next possible completion for 'text'.

If a command has not been entered, then complete against command list. Otherwise try to call complete_<command> to get list of completions.

complete_help (**args*)

completedefault (**ignored*)

Method called to complete an input line when no command-specific complete_*() method is available.

By default, it returns an empty list.

completenames (*text*, **ignored*)

default (*line*)

defaultFile ()

Produce a reasonable default.

dispatch_call (*frame*, *arg*)

dispatch_exception (*frame*, *arg*)

dispatch_line (*frame*)

dispatch_return (*frame*, *arg*)

do_EOF (*arg*)

do_a (*arg*)

do_alias (*arg*)

do_args (*arg*)

do_b (*arg*, *temporary=0*)

do_break (*arg*, *temporary=0*)

do_bt (*arg*)

do_c (*arg*)

do_cl (*arg*)

Three possibilities, tried in this order: clear -> clear all breaks, ask for confirmation clear file:lineno -> clear all breaks at file:lineno clear bpno bpno ... -> clear breakpoints by number

do_clear (*arg*)

Three possibilities, tried in this order: clear -> clear all breaks, ask for confirmation clear file:lineno -> clear all breaks at file:lineno clear bpno bpno ... -> clear breakpoints by number

do_commands (*arg*)

Defines a list of commands associated to a breakpoint Those commands will be executed whenever the breakpoint causes the program to stop execution.

do_condition (*arg*)

do_cont (*arg*)

do_continue (*arg*)

do_d (**args*, ***kw*)

do_debug (*arg*)

do_disable (*arg*)

do_down (**args*, ***kw*)

do_enable (*arg*)

do_exit (*arg*)

do_h (*arg*)

do_help (*arg*)

do_ignore (*arg*)

arg is bp number followed by ignore count.

do_j (*arg*)

do_jump (*arg*)

do_l (*arg*)

do_list (*arg*)

do_n (*arg*)

do_next (*arg*)

do_p (*arg*)

do_pdef (*arg*)

The debugger interface to magic_pdef

do_pdoc (*arg*)

The debugger interface to magic_pdoc

do_pinfo (*arg*)

The debugger equivalent of ?obj

do_pp (*arg*)

do_q (**args, **kw*)

do_quit (**args, **kw*)

do_r (*arg*)

do_restart (*arg*)

Restart program by raising an exception to be caught in the main debugger loop. If arguments were given, set them in sys.argv.

do_return (*arg*)

do_retval (*arg*)

do_run (*arg*)

Restart program by raising an exception to be caught in the main debugger loop. If arguments were given, set them in sys.argv.

do_rv (*arg*)

do_s (*arg*)

do_step (*arg*)

do_tbreak (*arg*)

do_u (**args, **kw*)

do_unalias (*arg*)

do_unt (*arg*)

do_until (*arg*)

do_up (**args, **kw*)

do_w (*arg*)

do_what_is (*arg*)

do_where (*arg*)

emptyline ()

Called when an empty line is entered in response to the prompt.

If this method is not overridden, it repeats the last nonempty command entered.

execRcLines ()

`forget()`
`format_stack_entry(frame_lineno, lprefix=': ', context=3)`
`get_all_breaks()`
`get_break(filename, lineno)`
`get_breaks(filename, lineno)`
`get_file_breaks(filename)`
`get_names()`
`get_stack(f, t)`
`handle_command_def(line)`
 Handles one command line during command list definition.
`help_EOF()`
`help_a()`
`help_alias()`
`help_args()`
`help_b()`
`help_break()`
`help_bt()`
`help_c()`
`help_cl()`
`help_clear()`
`help_commands()`
`help_condition()`
`help_cont()`
`help_continue()`
`help_d()`
`help_debug()`
`help_disable()`
`help_down()`
`help_enable()`
`help_exec()`
`help_exit()`
`help_h()`

```
help_help()
help_ignore()
help_j()
help_jump()
help_l()
help_list()
help_n()
help_next()
help_p()
help_pdb()
help_pp()
help_q()
help_quit()
help_r()
help_restart()
help_return()
help_run()
help_s()
help_step()
help_tbreak()
help_u()
help_unalias()
help_unt()
help_until()
help_up()
help_w()
help_what()
help_where()
interaction(frame, traceback)
lineinfo(identifier)
list_command_pydb(arg)
    List command to use if we have a newer pydb installed
```

lookupmodule (*filename*)

Helper function for break/clear parsing – may be overridden.

lookupmodule() translates (possibly incomplete) file or module name into an absolute file name.

new_do_down (*arg*)

new_do_frame (*arg*)

new_do_quit (*arg*)

new_do_restart (*arg*)

Restart command. In the context of ipython this is exactly the same thing as ‘quit’.

new_do_up (*arg*)

onecmd (*line*)

Interpret the argument as though it had been typed in response to the prompt.

Checks whether this line is typed at the normal prompt or in a breakpoint command list definition.

parseline (*line*)

Parse the line into a command name and a string containing the arguments. Returns a tuple containing (command, args, line). ‘command’ and ‘args’ may be None if the line couldn’t be parsed.

postcmd (*stop*, *line*)

Hook method executed just after a command dispatch is finished.

postloop ()

precmd (*line*)

Handle alias expansion and ‘;;’ separator.

preloop ()

Hook method executed once when the cmdloop() method is called.

print_list_lines (*filename*, *first*, *last*)

The printing (as opposed to the parsing part of a ‘list’ command).

print_stack_entry (*frame_lineno*, *prompt_prefix*=‘n-> ‘, *context*=3)

print_stack_trace ()

print_topics (*header*, *cmds*, *cmdlen*, *maxcol*)

reset ()

run (*cmd*, *globals*=None, *locals*=None)

runcall (*func*, **args*, ***kws*)

runtcx (*cmd*, *globals*, *locals*)

runeval (*expr*, *globals*=None, *locals*=None)

set_break (*filename*, *lineno*, *temporary*=0, *cond*=None, *funcname*=None)

set_colors (*scheme*)

Shorthand access to the color table scheme selector method.

set_continue ()

set_next (*frame*)

Stop on the next line in or below the given frame.

set_quit ()

set_return (*frame*)

Stop when returning from the given frame.

set_step ()

Stop after one line of code.

set_trace (*frame=None*)

Start debugging from *frame*.

If frame is not specified, debugging starts from caller's frame.

set_until (*frame*)

Stop when the line with the line no greater than the current one is reached or when returning from current frame

setup (*f, t*)

stop_here (*frame*)

trace_dispatch (*frame, event, arg*)

user_call (*frame, argument_list*)

This method is called when there is the remote possibility that we ever need to stop in this function.

user_exception (*frame, exc_info*)

user_line (*frame*)

This function is called when we stop or break at this line.

user_return (*frame, return_value*)

This function is called when a return trap is set here.

Tracer

class IPython.core.debugger.**Tracer** (*colors=None*)

Bases: object

Class for local debugging, similar to pdb.set_trace.

Instances of this class, when called, behave like pdb.set_trace, but providing IPython's enhanced capabilities.

This is implemented as a class which must be initialized in your own code and not as a standalone function because we need to detect at runtime whether IPython is already active or not. That detec-

tion is done in the constructor, ensuring that this code plays nicely with a running IPython, while functioning acceptably (though with limitations) if outside of it.

`__init__ (colors=None)`

Create a local debugger instance.

Parameters

- *colors* (None): a string containing the name of the color scheme to

use, it must be one of IPython's valid color schemes. If not given, the function will default to the current IPython scheme when running inside IPython, and to 'NoColor' otherwise.

Usage example:

```
from IPython.core.debugger import Tracer; debug_here = Tracer()
```

... later in your code `debug_here()` # -> will open up the debugger at that point.

Once the debugger activates, you can use all of its regular commands to step through code, set breakpoints, etc. See the `pdb` documentation from the Python standard library for usage details.

8.10.3 Functions

`IPython.core.debugger.BdbQuit_IPython_excepthook (self, et, ev, tb)`

`IPython.core.debugger.BdbQuit_excepthook (et, ev, tb)`

`IPython.core.debugger.decorate_fn_with_doc (new_fn, old_fn, additional_text='')`

Make `new_fn` have `old_fn`'s doc string. This is particularly useful for the **do_...** commands that hook into the help system. Adapted from from a `comp.lang.python` posting by Duncan Booth.

8.11 core.display

8.11.1 Module: `core.display`

Top-level display functions for displaying object in different formats.

Authors:

- Brian Granger

8.11.2 Functions

`IPython.core.display.display (*objs, **kwargs)`

Display a Python object in all frontends.

By default all representations will be computed and sent to the frontends. Frontends can decide which representation is used and how.

Parameters `objs` : tuple of objects

The Python objects to display.

include : list or tuple, optional

A list of format type strings (MIME types) to include in the format data dict. If this is set *only* the format types included in this list will be computed.

exclude : list or tuple, optional

A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

`IPython.core.display.display_html (*objs)`

Display the HTML representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

`IPython.core.display.display_json (*objs)`

Display the JSON representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

`IPython.core.display.display_latex (*objs)`

Display the LaTeX representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

`IPython.core.display.display_png (*objs)`

Display the PNG representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

`IPython.core.display.display_pretty (*objs)`

Display the pretty (default) representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

`IPython.core.display.display_svg (*objs)`

Display the SVG representation of an object.

Parameters **objs** : tuple of objects

The Python objects to display.

8.12 core.display_trap

8.12.1 Module: core.display_trap

Inheritance diagram for `IPython.core.display_trap`:



A context manager for handling `sys.displayhook`.

Authors:

- Robert Kern
- Brian Granger

8.12.2 DisplayTrap

class `IPython.core.display_trap.DisplayTrap` (*hook=None*)

Bases: `IPython.config.configurable.Configurable`

Object to manage `sys.displayhook`.

This came from `IPython.core.kernel.display_hook`, but is simplified (no callbacks or formatters) until more of the core is refactored.

__init__ (*hook=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

hook

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait `'a'`, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

set ()

Set the hook.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

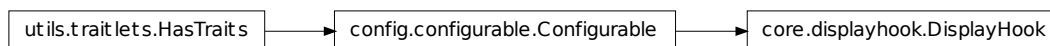
unset ()

Unset the hook.

8.13 core.displayhook

8.13.1 Module: core.displayhook

Inheritance diagram for IPython.core.displayhook:



Displayhook for IPython.

This defines a callable class that IPython uses for *sys.displayhook*.

Authors:

- Fernando Perez
- Brian Granger
- Robert Kern

8.13.2 DisplayHook

```
class IPython.core.displayhook.DisplayHook (shell=None, cache_size=1000, colors='NoColor', input_sep='n', output_sep='n', output_sep2=' ', ps1=None, ps2=None, ps_out=None, pad_left=True, config=None)
```

Bases: IPython.config.configurable.Configurable

The custom IPython displayhook to replace sys.displayhook.

This class does many things, but the basic idea is that it is a callable that gets called anytime user code returns a value.

Currently this class does more than just the displayhook logic and that extra logic should eventually be moved out of here.

```
__init__ (shell=None, cache_size=1000, colors='NoColor', input_sep='n', output_sep='n', output_sep2=' ', ps1=None, ps2=None, ps_out=None, pad_left=True, config=None)
```

```
check_for_underscore ()
```

Check if the user has set the ‘_’ variable by hand.

```
compute_format_data (result)
```

Compute format data of the object to be displayed.

The format data is a generalization of the `repr()` of an object. In the default implementation the format data is a `dict` of key value pair where the keys are valid MIME types and the values are JSON’able data structure containing the raw data for that MIME type. It is up to frontends to determine pick a MIME to use and display that data in an appropriate manner.

This method only computes the format data for the object and should NOT actually print or write that to a stream.

Parameters **result** : object

The Python object passed to the display hook, whose format will be computed.

Returns **format_data** : dict

A `dict` whose keys are valid MIME types and values are JSON’able raw data for that MIME type. It is recommended that all return values of this should always include the “text/plain” MIME type representation of the object.

```
config
```

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

```
finish_displayhook ()
```

Finish up all displayhook activities.

```
flush ()
```

log_output (*result*)

Log the output.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prompt_count

quiet ()

Should we silence the display hook because of ‘;’?

set_colors (*colors*)

Set the active color scheme and configure colors for the three prompt subsystems.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

start_displayhook ()

Start the displayhook, initializing resources.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

update_user_ns (*result*)

Update user_ns with various things like `_`, `__`, `_1`, etc.

write_format_data (*format_dict*)

Write the format data dict to the frontend.

This default version of this method simply writes the plain text representation of the object to `io.Term.cout`. Subclasses should override this method to send the entire *format_dict* to the frontends.

Parameters *format_dict* : dict

The format dict for the object passed to `sys.displayhook`.

write_output_prompt ()

Write the output prompt.

The default implementation simply writes the prompt to `io.Term.cout`.

8.14 core.displaypub

8.14.1 Module: core.displaypub

Inheritance diagram for `IPython.core.displaypub`:



An interface for publishing rich data to frontends.

There are two components of the display system:

- Display formatters, which take a Python object and compute the representation of the object in various formats (text, HTML, SVG, etc.).
- The display publisher that is used to send the representation data to the various frontends.

This module defines the logic display publishing. The display publisher uses the `display_data` message type that is defined in the IPython messaging spec.

Authors:

- Brian Granger

8.14.2 DisplayPublisher

class `IPython.core.displaypub.DisplayPublisher` (***kwargs*)

Bases: `IPython.config.configurable.Configurable`

A traitled class that publishes display data to frontends.

Instances of this class are created by the main IPython object and should be accessed there.

__init__ (**kwargs)

Create a configurable given a config config.

Parameters **config** : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

publish (source, data, metadata=None)

Publish data and metadata to all frontends.

See the `display_data` message in the messaging documentation for more details about this message type.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/latex
- application/json
- image/png
- image/svg+xml

Parameters `source` : str

A string that give the function or method that created the data, such as 'IPython.core.page'.

data : dict

A dictionary having keys that are valid MIME types (like 'text/plain' or 'image/svg+xml') and values that are the data for that MIME type. The data itself must be a JSON'able data structure. Minimally all data should have the 'text/plain' data, which can be displayed by all frontends. If more than the plain text is given, it is up to the frontend to decide which representation to use.

metadata : dict

A dictionary for metadata related to the data. This can contain arbitrary key, value pairs that frontends can use to interpret the data.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

```
IPython.core.displaypub.publish_display_data(self, source, data, meta-
                                              data=None)
```

Publish data and metadata to all frontends.

See the `display_data` message in the messaging documentation for more details about this message type.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/latex
- application/json
- image/png
- image/svg+xml

Parameters `source` : str

A string that give the function or method that created the data, such as `'IPython.core.page'`.

data : dict

A dictionary having keys that are valid MIME types (like `'text/plain'` or `'image/svg+xml'`) and values that are the data for that MIME type. The data itself must be a JSON'able data structure. Minimally all data should have the `'text/plain'` data, which can be displayed by all frontends. If more than the plain text is given, it is up to the frontend to decide which representation to use.

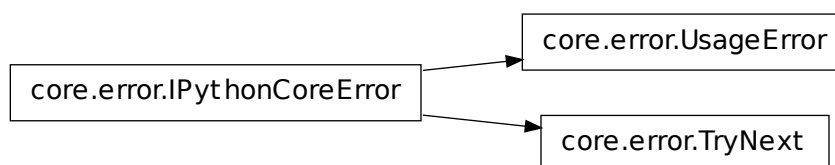
metadata : dict

A dictionary for metadata related to the data. This can contain arbitrary key, value pairs that frontends can use to interpret the data.

8.15 core.error

8.15.1 Module: `core.error`

Inheritance diagram for `IPython.core.error`:



Global exception classes for IPython.core.

Authors:

- Brian Granger
- Fernando Perez

Notes

8.15.2 Classes

`IPythonCoreError`

class `IPython.core.error.IPythonCoreError`

Bases: `exceptions.Exception`

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`args`

`message`

`TryNext`

class `IPython.core.error.TryNext (*args, **kwargs)`

Bases: `IPython.core.error.IPythonCoreError`

Try next hook exception.

Raise this in your hook function to indicate that the next hook handler should be used to handle the operation. If you pass arguments to the constructor those arguments will be used by the next hook instead of the original ones.

`__init__ (*args, **kwargs)`

`args`

`message`

`UsageError`

class `IPython.core.error.UsageError`

Bases: `IPython.core.error.IPythonCoreError`

Error in magic function arguments, etc.

Something that probably won't warrant a full traceback, but should nevertheless interrupt a macro / batch file.

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

args
message

8.16 core.excolors

8.16.1 Module: `core.excolors`

Color schemes for exception handling code in IPython.

`IPython.core.excolors.exception_colors()`
Return a color table with fields for exception reporting.

The table is an instance of `ColorSchemeTable` with schemes added for ‘Linux’, ‘LightBG’ and ‘No-Color’ and fields for exception handling filled in.

Examples:

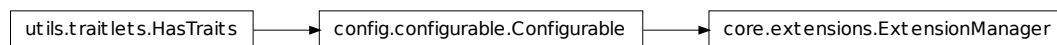
```
>>> ec = exception_colors()
>>> ec.active_scheme_name
''
>>> print ec.active_colors
None
```

Now we activate a color scheme: `>>> ec.set_active_scheme('NoColor') >>> ec.active_scheme_name` ‘NoColor’ `>>> ec.active_colors.keys()` [‘em’, ‘filenameEm’, ‘excName’, ‘valEm’, ‘nameEm’, ‘line’, ‘topline’, ‘name’, ‘caret’, ‘val’, ‘vName’, ‘Normal’, ‘filename’, ‘linenoEm’, ‘lineno’, ‘normalEm’]

8.17 core.extensions

8.17.1 Module: `core.extensions`

Inheritance diagram for `IPython.core.extensions`:



A class for managing IPython extensions.

Authors:

- Brian Granger

8.17.2 ExtensionManager

class IPython.core.extensions.**ExtensionManager** (*shell=None, config=None*)

Bases: IPython.config.configurable.Configurable

A class to manage IPython extensions.

An IPython extension is an importable Python module that has a function with the signature:

```
def load_ipython_extension(ipython):
    # Do things with ipython
```

This function is called after your extension is imported and the currently active InteractiveShell instance is passed as the only argument. You can do anything you want with IPython at that point, including defining new magic and aliases, adding new components, etc.

The `load_ipython_extension()` will be called again is you load or reload the extension again. It is up to the extension author to add code to manage that.

You can put your extension modules anywhere you want, as long as they can be imported by Python's standard import mechanism. However, to make it easy to write extensions, you can also put your extensions in `os.path.join(self.ipython_dir, 'extensions')`. This directory is added to `sys.path` automatically.

__init__ (*shell=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

ipython_extension_dir

load_extension (*module_str*)

Load an IPython extension by its module name.

If `load_ipython_extension()` returns anything, this function will return that object.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

reload_extension (*module_str*)

Reload an IPython extension by calling reload.

If the module has not been loaded before, `InteractiveShell.load_extension()` is called. Otherwise `reload()` is called and then the `load_ipython_extension()` function of the module, if it exists is called.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

unload_extension (*module_str*)

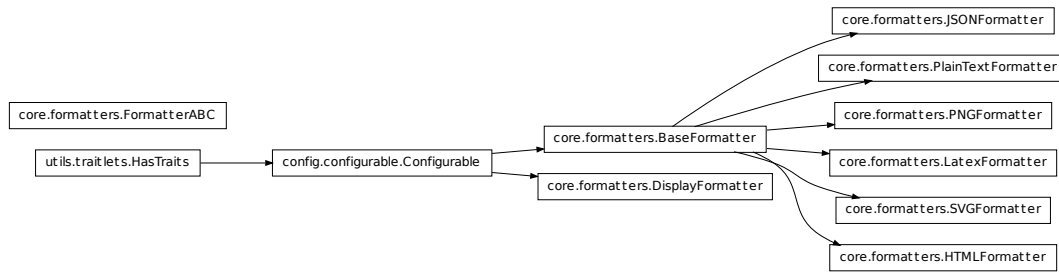
Unload an IPython extension by its module name.

This function looks up the extension's name in `sys.modules` and simply calls `mod.unload_ipython_extension(self)`.

8.18 core.formatters

8.18.1 Module: `core.formatters`

Inheritance diagram for `IPython.core.formatters`:



Display formatters.

Authors:

- Robert Kern
- Brian Granger

8.18.2 Classes

BaseFormatter

class IPython.core.formatters.**BaseFormatter** (**kwargs)
 Bases: IPython.config.configurable.Configurable

A base formatter class that is configurable.

This formatter should usually be used as the base class of all formatters. It is a traitled `Configurable` class and includes an extensible API for users to determine how their objects are formatted. The following logic is used to find a function to format an given object.

1. The object is introspected to see if it has a method with the name `print_method`. If it does, that object is passed to that method for formatting.
2. If no print method is found, three internal dictionaries are consulted to find print method: `singleton_printers`, `type_printers` and `deferred_printers`.

Users should use these dictionaries to register functions that will be used to compute the format data for their objects (if those objects don't have the special print methods). The easiest way of using these dictionaries is through the `for_type()` and `for_type_by_name()` methods.

If no function/callable is found to compute the format data, `None` is returned and this format type is not used.

__init__ (**kwargs)
 Create a configurable given a config config.

Parameters `config`: Config

If this is empty, default values are used. If `config` is a `Config` instance, it will be used to configure the instance.

Notes

Subclasses of `Configurable` must call the `__init__()` method of `Configurable` *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the

raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

type_printers

An instance of a Python dict.

DisplayFormatter

class IPython.core.formatters.**DisplayFormatter**(**kwargs)

Bases: IPython.config.configurable.Configurable

__init__(**kwargs)

Create a configurable given a config config.

Parameters **config** : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the **__init__()** method of Configurable *before* doing anything else and using **super()**:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

format(obj, include=None, exclude=None)

Return a format data dict for an object.

By default all format types will be computed.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/latex
- application/json
- image/png
- image/svg+xml

Parameters **obj** : object

The Python object whose format data will be computed.

include : list or tuple, optional

A list of format type strings (MIME types) to include in the format data dict. If this is set *only* the format types included in this list will be computed.

exclude : list or tuple, optional

A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

Returns **format_dict** : dict

A dictionary of key/value pairs, one for each format that was generated for the object. The keys are the format types, which will usually be MIME type strings and the values are JSON-able data structures containing the raw data for the representation in that format.

format_types

Return the format types (MIME types) of the active formatters.

formatters

An instance of a Python dict.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

plain_text_only

A boolean (True, False) trait.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this class's traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

FormatterABC

```
class IPython.core.formatters.FormatterABC
```

Bases: object

Abstract base class for Formatters.

A formatter is a callable class that is responsible for computing the raw format data for a particular format type (MIME type). For example, an HTML formatter would have a format type of *text/html* and would return the HTML representation of the object when called.

```
__init__()
```

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

HTMLFormatter

```
class IPython.core.formatters.HTMLFormatter(**kwargs)
```

Bases: `IPython.core.formatters.BaseFormatter`

An HTML formatter.

To define the callables that compute the HTML representation of your objects, define a `__html__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

```
__init__(**kwargs)
```

Create a configurable given a config config.

Parameters `config`: Config

If this is empty, default values are used. If `config` is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

JSONFormatter

class `IPython.core.formatters.JSONFormatter` (***kwargs*)

Bases: `IPython.core.formatters.BaseFormatter`

A JSON string formatter.

To define the callables that compute the JSON string representation of your objects, define a `__json__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

__init__ (***kwargs*)

Create a configurable given a config config.

Parameters **config** : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

type_printers

An instance of a Python dict.

LatexFormatter

```
class IPython.core.formatters.LatexFormatter(**kwargs)
```

```
    Bases: IPython.core.formatters.BaseFormatter
```

A LaTeX formatter.

To define the callables that compute the LaTeX representation of your objects, define a `__latex__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

`__init__` (***kwargs*)

Create a configurable given a config config.

Parameters `config` : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters `typ` : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters `type_module` : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘`a`’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The `TraitTypes` returned don’t know anything about the values that the various `HasTrait`’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

PNGFormatter

class `IPython.core.formatters.PNGFormatter(**kwargs)`

Bases: `IPython.core.formatters.BaseFormatter`

A PNG formatter.

To define the callables that compute the PNG representation of your objects, define a `__png__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this. The raw data should be the base64 encoded raw png data.

__init__ (***kwargs*)

Create a configurable given a config config.

Parameters **config** : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module*, *type_name*, *func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler*, *name=None*, *remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘`a`’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

PlainTextFormatter

class IPython.core.formatters.PlainTextFormatter (***kwargs*)

Bases: IPython.core.formatters.BaseFormatter

The default pretty-printer.

This uses IPython.external.pretty to compute the format data of the object. If the object cannot be pretty printed, repr() is used. See the documentation of IPython.external.pretty for details on how to write pretty printers. Here is a simple example:

```
def dtype_pprinter(obj, p, cycle):
    if cycle:
        return p.text('dtype(...)')
    if hasattr(obj, 'fields'):
        if obj.fields is None:
            p.text(repr(obj))
        else:
            p.begin_group(7, 'dtype([')
            for i, field in enumerate(obj.descr):
                if i > 0:
                    p.text(',')
                    p.breakable()
                p.pretty(field)
            p.end_group(7, '])')
```

__init__ (***kwargs*)

Create a configurable given a config config.

Parameters **config**: Config

If this is empty, default values are used. If `config` is a `Config` instance, it will be used to configure the instance.

Notes

Subclasses of `Configurable` must call the `__init__()` method of `Configurable` *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the

raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

max_width

A integer trait.

newline

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

pprint

A boolean (True, False) trait.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

verbose

A boolean (True, False) trait.

SVGFormatter

```
class IPython.core.formatters.SVGFormatter(**kwargs)
```

Bases: `IPython.core.formatters.BaseFormatter`

An SVG formatter.

To define the callables that compute the SVG representation of your objects, define a `__svg__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

```
__init__(**kwargs)
```

Create a configurable given a config config.

Parameters `config`: `Config`

If this is empty, default values are used. If `config` is a `Config` instance, it will be used to configure the instance.

Notes

Subclasses of `Configurable` must call the `__init__()` method of `Configurable` *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

for_type (*typ, func*)

Add a format function for a given type.

Parameters **typ** : class

The class of the object that will be formatted using *func*.

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

for_type_by_name (*type_module, type_name, func*)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters **type_module** : str

The full dotted name of the module the type is defined in, like `numpy`.

type_name : str

The name of the type (the class name), like `dtype`

func : callable

The callable that will be called to compute the format data. The call signature of this function is simple, it must take the object to be formatted and return the raw data for the given format. Subclasses may use a different call signature for the *func* argument.

format_type

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘`a`’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

print_method

A trait for strings.

singleton_printers

An instance of a Python dict.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

8.18.3 Function

IPython.core.formatters.**format_display_data** (*obj*, *include=None*, *exclude=None*)

Return a format data dict for an object.

By default all format types will be computed.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/latex
- application/json
- image/png
- image/svg+xml

Parameters *obj*: object

The Python object whose format data will be computed.

Returns *format_dict*: dict

A dictionary of key/value pairs, one or each format that was generated for the object. The keys are the format types, which will usually be MIME type

strings and the values and JSON'able data structure containing the raw data for the representation in that format.

include : list or tuple, optional

A list of format type strings (MIME types) to include in the format data dict. If this is set *only* the format types included in this list will be computed.

exclude : list or tuple, optional

A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

8.19 core.history

8.19.1 Module: `core.history`

Inheritance diagram for `IPython.core.history`:



History related magics and functionality

8.19.2 Classes

`HistoryManager`

class `IPython.core.history.HistoryManager` (*shell*)

Bases: `object`

A class to organize all history-related functionality in one place.

__init__ (*shell*)

Create a new history manager associated with a shell instance.

autosave_if_due ()

Check if the autosave event is set; if so, save history. We do it this way so that the save takes place in the main thread.

get_history (*index=None, raw=False, output=True*)

Get the history list.

Get the input and output history.

Parameters **index** : n or (n1, n2) or None

If n, then the last entries. If a tuple, then all in range(n1, n2). If None, then all entries. Raises IndexError if the format of index is incorrect.

raw : bool

If True, return the raw input.

output : bool

If True, then return the output as well.

Returns If output is True, then return a dict of tuples, keyed by the prompt :

numbers and with values of (input, output). If output is False, then :

a dict, keyed by the prompt number with the values of input. Raises :

IndexError if no history is found. :

populate_readline_history ()

Populate the readline history from the raw history.

We only store one copy of the raw history, which is persisted to a json file on disk. The readline history is repopulated from the contents of this file.

reload_history ()

Reload the input history from disk file.

reset ()

Clear all histories managed by this object.

save_history ()

Save input history to a file (via readline library).

store_inputs (*source, source_raw=None*)

Store source and raw input in history and create input cache variables `_i*`.

Parameters **source** : str

Python input.

source_raw : str, optional

If given, this is the raw input without any IPython transformations applied to it. If not given, `source` is used.

sync_inputs ()

Ensure raw and translated histories have same length.

HistorySaveThread

```
class IPython.core.history.HistorySaveThread (autosave_flag, time_interval=60)
```

Bases: threading.Thread

This thread makes IPython save history periodically.

Without this class, IPython would only save the history on a clean exit. This saves the history periodically (the current default is once per minute), so that it is not lost in the event of a crash.

The implementation sets an event to indicate that history should be saved. The actual save is carried out after executing a user command, to avoid thread issues.

```
__init__ (autosave_flag, time_interval=60)
```

```
getName ()
```

```
ident
```

```
isAlive ()
```

```
isDaemon ()
```

```
is_alive ()
```

```
join (timeout=None)
```

```
name
```

```
run ()
```

```
setDaemon (daemonic)
```

```
setName (name)
```

```
start ()
```

```
stop ()
```

Safely and quickly stop the autosave timer thread.

ShadowHist

```
class IPython.core.history.ShadowHist (db, shell)
```

Bases: object

```
__init__ (db, shell)
```

```
add (ent)
```

```
all ()
```

```
get (idx)
```

```
inc_idx ()
```

8.19.3 Functions

`IPython.core.history.init_ipython(ip)`

`IPython.core.history.magic_hist(self, parameter_s='')`

Alternate name for `%history`.

`IPython.core.history.magic_history(self, parameter_s='')`

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

`%history` -> print at most 40 inputs (some may be multi-line)
`%history n` -> print at most `n` inputs
`%history n1 n2` -> print inputs between `n1` and `n2` (`n2` not included)

By default, input history is printed without line numbers so it can be directly pasted into an editor.

With `-n`, each input's number `<n>` is shown, and is accessible as the automatically generated variable `_i<n>` as well as `In[<n>]`. Multi-line statements are printed starting at a new line for easy copy/paste.

Options:

- n: print line numbers for each input. This feature is only available if numbered prompts are in use.

- o: also print outputs for each input.

- p: **print classic '>>>' python prompts before each input. This is useful** for making documentation, and in conjunction with `-o`, for producing doctest-ready output.

- r: (default) print the 'raw' history, i.e. the actual commands you typed.

- t: print the 'translated' history, as IPython understands it. IPython filters your input and converts it all into valid Python source before executing it (things like magics or aliases are turned into function calls, for example). With this option, you'll see the native history instead of the user-entered version: `'%cd /'` will be seen as `'get_ipython().magic("%cd /")'` instead of `'%cd /'`.

- g: treat the arg as a pattern to grep for in (full) history. This includes the "shadow history" (almost all commands ever written). Use `'%hist -g'` to show full shadow history (may be very long). In shadow history, every index number starts with 0.

- f **FILENAME: instead of printing the output to the screen, redirect it to** the given file. The file is always overwritten, though IPython asks for confirmation first if it already exists.

`IPython.core.history.rep_f(self, arg)`

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

- `%rep` (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:

```
$ l = ["hei", "vaan"]
$ "".join(l)
==> heivaan
```

```
$ %rep
$ heivaan_ <== cursor blinking
```

```
%rep 45
```

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

```
%rep 1-4 6-7 3
```

Repeat the specified lines immediately. Input slice syntax is the same as in `%macro` and `%save`.

```
%rep foo
```

Place the most recent line that has the substring “foo” to next input. (e.g. ‘svn ci -m foobar’).

8.20 core.hooks

8.20.1 Module: `core.hooks`

Inheritance diagram for `IPython.core.hooks`:

`core.hooks.CommandChainDispatcher`

hooks for IPython.

In Python, it is possible to overwrite any method of any object if you really want to. But IPython exposes a few ‘hooks’, methods which are `_designed_` to be overwritten by users for customization purposes. This module defines the default versions of all such hooks, which get used by IPython if not overridden by the user.

hooks are simple functions, but they should be declared with ‘self’ as their first argument, because when activated they are registered into IPython as instance methods. The self argument will be the IPython running instance itself, so hooks have full access to the entire IPython object.

If you wish to define a new hook and activate it, you need to put the necessary code into a python file which can be either imported or `execfile()`’d from within your `ipythonrc` configuration.

For example, suppose that you have a module called ‘myiphooks’ in your `PYTHONPATH`, which contains the following definition:

```
import os from IPython.core import ipapi ip = ipapi.get()
```

```
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()
```

```
ip.set_hook('editor', calljed)
```

You can then enable the functionality by doing ‘import myiphooks’ somewhere in your configuration files or ipython command line.

8.20.2 Class

8.20.3 CommandChainDispatcher

class IPython.core.hooks.**CommandChainDispatcher** (*commands=None*)

Dispatch calls to a chain of commands until some func can handle it

Usage: instantiate, execute “add” to add commands (with optional priority), execute normally via f() calling mechanism.

__init__ (*commands=None*)

add (*func, priority=0*)

Add a func to the cmd chain with given priority

8.20.4 Functions

IPython.core.hooks.**clipboard_get** (*self*)

Get text from the clipboard.

IPython.core.hooks.**editor** (*self, filename, linenum=None*)

Open the default editor at the given filename and linenum.

This is IPython’s default editor hook, you can use it as an example to write your own modified one. To set your own editor function as the new editor hook, call ip.set_hook(‘editor’,yourfunc).

IPython.core.hooks.**fix_error_editor** (*self, filename, linenum, column, msg*)

Open the editor at the given filename, linenum, column and show an error message. This is used for correcting syntax errors. The current implementation only has special support for the VIM editor, and falls back on the ‘editor’ hook if VIM is not used.

Call ip.set_hook(‘fix_error_editor’,yourfunc) to use your own function,

IPython.core.hooks.**generate_prompt** (*self, is_continuation*)

calculate and return a string with the prompt to display

IPython.core.hooks.**input_prefilter** (*self, line*)

Default input prefilter

This returns the line as unchanged, so that the interpreter knows that nothing was done and proceeds with “classic” prefiltering (%magics, !shell commands etc.).

Note that leading whitespace is not passed to this hook. Prefilter can’t alter indentation.

IPython.core.hooks.**late_startup_hook** (*self*)

Executed after ipython has been constructed and configured

`IPython.core.hooks.pre_prompt_hook(self)`

Run before displaying the next prompt

Use this e.g. to display output from asynchronous operations (in order to not mess up text entry)

`IPython.core.hooks.pre_run_code_hook(self)`

Executed before running the (prefiltered) code in IPython

`IPython.core.hooks.show_in_pager(self, s)`

Run a string through pager

`IPython.core.hooks.shutdown_hook(self)`

default shutdown hook

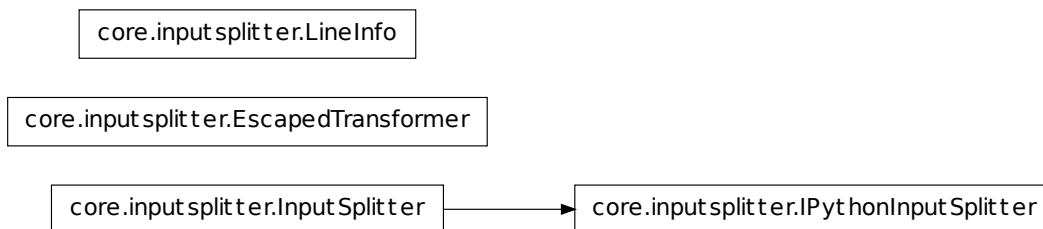
Typically, shutdown hooks should raise `TryNext` so all shutdown ops are done

`IPython.core.hooks.synchronize_with_editor(self, filename, linenum, column)`

8.21 core.inputsplitter

8.21.1 Module: `core.inputsplitter`

Inheritance diagram for `IPython.core.inputsplitter`:



Analysis of text input into executable blocks.

The main class in this module, `InputSplitter`, is designed to break input from either interactive, line-by-line environments or block-based ones, into standalone blocks that can be executed by Python as ‘single’ statements (thus triggering `sys.displayhook`).

A companion, `IPythonInputSplitter`, provides the same functionality but with full support for the extended IPython syntax (magics, system calls, etc).

For more details, see the class docstring below.

Syntax Transformations

One of the main jobs of the code in this file is to apply all syntax transformations that make up ‘the IPython language’, i.e. magics, shell escapes, etc. All transformations should be implemented as *fully stateless*

entities, that simply take one line as their input and return a line. Internally for implementation purposes they may be a normal function or a callable object, but the only input they receive will be a single line and they should only return a line, without holding any data-dependent state between calls.

As an example, the `EscapedTransformer` is a class so we can more clearly group together the functionality of dispatching to individual functions based on the starting escape character, but the only method for public use is its `call` method.

ToDo

- Should we make `push()` actually raise an exception once `push_accepts_more()` returns `False`?
- Naming cleanups. The `tr_*` names aren't the most elegant, though now they are at least just attributes of a class so not really very exposed.
- Think about the best way to support dynamic things: automagic, autocall, macros, etc.
- Think of a better heuristic for the application of the transforms in `IPythonInputSplitter.push()` than looking at the buffer ending in `'.'`. Idea: track indentation change events (indent, dedent, nothing) and apply them only if the indentation went up, but not otherwise.
- Think of the cleanest way for supporting user-specified transformations (the user prefilters we had before).

Authors

- Fernando Perez
- Brian Granger

8.21.2 Classes

`EscapedTransformer`

```
class IPython.core.inputsplitter.EscapedTransformer
```

```
    Bases: object
```

```
    Class to transform lines that are explicitly escaped out.
```

```
    __init__()
```

`IPythonInputSplitter`

```
class IPython.core.inputsplitter.IPythonInputSplitter (input_mode=None)
```

```
    Bases: IPython.core.inputsplitter.InputSplitter
```

```
    An input splitter that recognizes all of IPython's special syntax.
```

```
    __init__(input_mode=None)
```

push (*lines*)

Push one or more lines of IPython input.

push_accepts_more ()

Return whether a block of interactive input can accept more input.

This method is meant to be used by line-oriented frontends, who need to guess whether a block is complete or not based solely on prior and current input lines. The InputSplitter considers it has a complete interactive block and will not accept more input only when either a SyntaxError is raised, or *all* of the following are true:

- 1.The input compiles to a complete statement.
- 2.The indentation level is flush-left (because if we are indented, like inside a function definition or for loop, we need to keep reading new input).
- 3.There is one extra line consisting only of whitespace.

Because of condition #3, this method should be used only by *line-oriented* frontends, since it means that intermediate blank lines are not allowed in function definitions (or any other indented block).

Block-oriented frontends that have a separate keyboard event to indicate execution should use the `split_blocks()` method instead.

If the current input produces a syntax error, this method immediately returns False but does *not* raise the syntax error exception, as typically clients will want to send invalid syntax to an execution backend which might convert the invalid syntax into valid Python via one of the dynamic IPython mechanisms.

reset ()

Reset the input buffer and associated state.

source_raw_reset ()

Return input and raw source and perform a full reset.

source_reset ()

Return the input source and perform a full reset.

split_blocks (*lines*)

Split a multiline string into multiple input blocks.

Note: this method starts by performing a full reset().

Parameters **lines** : str

A possibly multiline string.

Returns **blocks** : list

A list of strings, each possibly multiline. Each string corresponds to a single block that can be compiled in ‘single’ mode (unless it has a syntax error).

InputSplitter

class IPython.core.inputsplitter.**InputSplitter** (*input_mode=None*)

Bases: object

An object that can split Python source input in executable blocks.

This object is designed to be used in one of two basic modes:

- 1.By feeding it python source line-by-line, using `push()`. In this mode, it will return on each push whether the currently pushed code could be executed already. In addition, it provides a method called `push_accepts_more()` that can be used to query whether more input can be pushed into a single interactive block.
- 2.By calling `split_blocks()` with a single, multiline Python string, that is then split into blocks each of which can be executed interactively as a single statement.

This is a simple example of how an interactive terminal-based client can use this tool:

```
isp = InputSplitter()
while isp.push_accepts_more():
    indent = ' '*isp.indent_spaces
    prompt = '>>> ' + indent
    line = indent + raw_input(prompt)
    isp.push(line)
print 'Input source was:'
```

`, isp.source_reset(),`

__init__ (*input_mode=None*)

Create a new InputSplitter instance.

Parameters `input_mode` : str

One of ['line', 'cell']; default is 'line'.

The `input_mode` parameter controls how new inputs are used when fed via :

the `:meth:'push'` method: :

- **'line':** meant for line-oriented clients, inputs are appended one at a :

time to the internal buffer and the whole buffer is compiled.

- **'cell':** meant for clients that can edit multi-line 'cells' of text at :

a time. A cell can contain one or more blocks that can be compile in 'single' mode by Python. In this mode, each new input new input completely replaces all prior inputs. Cell mode is thus equivalent to prepending a full `reset()` to every `push()` call.

push (*lines*)

Push one or more lines of input.

This stores the given lines and returns a status code indicating whether the code forms a complete Python block or not.

Any exceptions generated in compilation are swallowed, but if an exception was produced, the method returns True.

Parameters `lines` : string

One or more lines of Python input.

Returns `is_complete` : boolean

True if the current input source (the result of the current input

plus prior inputs) forms a complete Python execution block. Note that :

this value is also stored as a private attribute (`_is_complete`), so it :

can be queried at any time. :

`push_accepts_more()`

Return whether a block of interactive input can accept more input.

This method is meant to be used by line-oriented frontends, who need to guess whether a block is complete or not based solely on prior and current input lines. The InputSplitter considers it has a complete interactive block and will not accept more input only when either a SyntaxError is raised, or *all* of the following are true:

- 1.The input compiles to a complete statement.
- 2.The indentation level is flush-left (because if we are indented, like inside a function definition or for loop, we need to keep reading new input).
- 3.There is one extra line consisting only of whitespace.

Because of condition #3, this method should be used only by *line-oriented* frontends, since it means that intermediate blank lines are not allowed in function definitions (or any other indented block).

Block-oriented frontends that have a separate keyboard event to indicate execution should use the `split_blocks()` method instead.

If the current input produces a syntax error, this method immediately returns False but does *not* raise the syntax error exception, as typically clients will want to send invalid syntax to an execution backend which might convert the invalid syntax into valid Python via one of the dynamic IPython mechanisms.

`reset()`

Reset the input buffer and associated state.

`source_reset()`

Return the input source and perform a full reset.

`split_blocks(lines)`

Split a multiline string into multiple input blocks.

Note: this method starts by performing a full reset().

Parameters `lines` : str

A possibly multiline string.

Returns `blocks` : list

A list of strings, each possibly multiline. Each string corresponds to a single block that can be compiled in ‘single’ mode (unless it has a syntax error).

LineInfo

class `IPython.core.inputsplitter.LineInfo` (*line*)

Bases: object

A single line of input and associated info.

This is a utility class that mostly wraps the output of `split_user_input()` into a convenient object to be passed around during input transformations.

Includes the following as properties:

line The original, raw line

lspace Any early whitespace before actual text starts.

esc The initial esc character (or characters, for double-char escapes like ‘??’ or ‘!!’).

fpart The ‘function part’, which is basically the maximal initial sequence of valid python identifiers and the ‘.’ character. This is what is checked for alias and magic transformations, used for auto-calling, etc.

rest Everything else on the line.

`__init__` (*line*)

8.21.3 Functions

`IPython.core.inputsplitter.get_input_encoding()`

Return the default standard input encoding.

If sys.stdin has no encoding, ‘ascii’ is returned.

`IPython.core.inputsplitter.num_ini_spaces(s)`

Return the number of initial spaces in a string.

Note that tabs are counted as a single space. For now, we do *not* support mixing of tabs and spaces in the user’s input.

Parameters `s` : string

Returns `n` : int

`IPython.core.inputsplitter.remove_comments(src)`

Remove all comments from input source.

Note: comments are NOT recognized inside of strings!

Parameters `src` : string

A single or multiline input string.

Returns String with all Python comments removed. :

IPython.core.inputsplitter.**split_blocks** (*python*)

Split multiple lines of code into discrete commands that can be executed singly.

Parameters *python* : str

Pure, exec'able Python code.

Returns *commands* : list of str

Separate commands that can be exec'ed independently.

IPython.core.inputsplitter.**split_user_input** (*line*)

Split user input into early whitespace, esc-char, function part and rest.

This is currently handles lines with '=' in them in a very inconsistent manner.

Examples

```
>>> split_user_input('x=1')
(' ', ' ', 'x=1', ' ')
>>> split_user_input('?')
(' ', '?', ' ', ' ')
>>> split_user_input('??')
(' ', '??', ' ', ' ')
>>> split_user_input(' ?')
(' ', '?', ' ', ' ')
>>> split_user_input(' ??')
(' ', '??', ' ', ' ')
>>> split_user_input('??x')
(' ', '??', 'x', ' ')
>>> split_user_input('?x=1')
(' ', ' ', '?x=1', ' ')
>>> split_user_input('!ls')
(' ', '!', 'ls', ' ')
>>> split_user_input(' !ls')
(' ', ' ', '!', 'ls', ' ')
>>> split_user_input('!!ls')
(' ', '!!', 'ls', ' ')
>>> split_user_input(' !!ls')
(' ', ' ', '!!', 'ls', ' ')
>>> split_user_input(',ls')
(' ', ',', 'ls', ' ')
>>> split_user_input(';ls')
(' ', ';', 'ls', ' ')
>>> split_user_input(' ;ls')
(' ', ' ', ';', 'ls', ' ')
>>> split_user_input('f.g(x)')
(' ', ' ', 'f.g(x)', ' ')
>>> split_user_input('f.g (x)')
(' ', ' ', 'f.g', ' (x)')
>>> split_user_input('?%hist')
(' ', '?', '%hist', ' ')
```

```
>>> split_user_input(' ?x*')
(' ', '?', 'x*', '')
```

IPython.core.inputsplitter.**transform_assign_magic**(*line*)

Handle the `a = %who` syntax.

IPython.core.inputsplitter.**transform_assign_system**(*line*)

Handle the `files = !ls` syntax.

IPython.core.inputsplitter.**transform_classic_prompt**(*line*)

Handle inputs that start with `'>>> '` syntax.

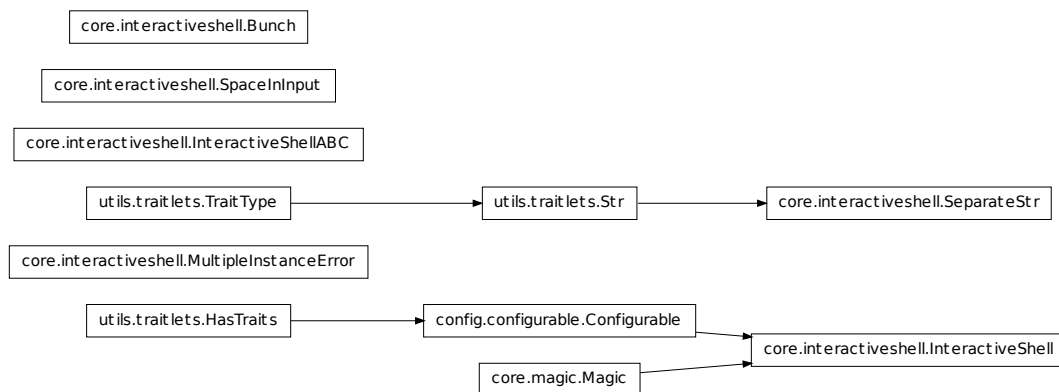
IPython.core.inputsplitter.**transform_ipy_prompt**(*line*)

Handle inputs that start classic IPython prompt syntax.

8.22 core.interactiveshell

8.22.1 Module: core.interactiveshell

Inheritance diagram for IPython.core.interactiveshell:



Main IPython class.

8.22.2 Classes

Bunch

class IPython.core.interactiveshell.**Bunch**

InteractiveShell

```
class IPython.core.interactiveshell.InteractiveShell (config=None,
                                                    ipython_dir=None,
                                                    user_ns=None,
                                                    user_global_ns=None,
                                                    custom_exceptions=(),
                                                    None))
```

Bases: IPython.config.configurable.Configurable, IPython.core.magic.Magic

An enhanced, interactive shell for Python.

```
__init__(config=None, ipython_dir=None, user_ns=None, user_global_ns=None, custom_exceptions=(), None))
```

alias_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

arg_err (func)

Print docstring if incorrect arguments were passed

ask_yes_no (prompt, default=True)

atexit_operations ()

This will be executed at the time of exit.

Cleanup operations and saving of persistent data that is done unconditionally by IPython should be performed here.

For things that may depend on startup flags or platform specifics (such as having readline or not), register a separate atexit function in the code that has the appropriate information, rather than trying to clutter

auto_rewrite_input (cmd)

Print to the screen the rewritten form of the user's command.

This shows visual feedback by rewriting input lines that cause automatic calling to kick in, like:

```
/f x
```

into:

```
-----> f(x)
```

after the user's input prompt. This helps the user understand that the input line was transformed automatically by IPython.

autocall

An enum that whose value must be in a given sequence.

autoindent

A casting version of the boolean trait.

automagic

A casting version of the boolean trait.

builtin_trap

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

cache_main_mod(*ns*, *fname*)

Cache a main module's namespace.

When scripts are executed via `%run`, we must keep a reference to the namespace of their `__main__` module (a `FakeModule` instance) around so that Python doesn't clear it, rendering objects defined therein useless.

This method keeps said reference in a private dict, keyed by the absolute path of the module object (which corresponds to the script path). This way, for multiple executions of the same script we only keep one copy of the namespace (the last one), thus preventing memory leaks from old references while allowing the objects from the last execution to be accessible.

Note: we can not allow the actual `FakeModule` instances to be deleted, because of how Python tears down modules (it hard-sets all their references to `None` without regard for reference counts). This method must therefore make a *copy* of the given namespace, to allow the original module's `__dict__` to be cleared and reused.

Parameters **ns** : a namespace (a dict, typically)

fname [str] Filename associated with the namespace.

Examples

```
In [10]: import IPython
```

```
In [11]: _ip.cache_main_mod(IPython.__dict__,IPython.__file__)
```

```
In [12]: IPython.__file__ in _ip._main_ns_cache Out[12]: True
```

cache_size

A integer trait.

call_pdb

Control auto-activation of `pdb` at exceptions

cleanup()**clear_main_mod_cache**()

Clear the cache of main modules.

Mainly for use by utilities like `%reset`.

Examples

```
In [15]: import IPython
```

```
In [16]: _ip.cache_main_mod(IPython.__dict__,IPython.__file__)
```

```
In [17]: len(_ip._main_ns_cache) > 0 Out[17]: True
```

```
In [18]: _ip.clear_main_mod_cache()
```

```
In [19]: len(_ip._main_ns_cache) == 0 Out[19]: True
```

color_info

A casting version of the boolean trait.

colors

An enum of strings that are caseless in validate.

complete (*text*, *line=None*, *cursor_pos=None*)

Return the completed text and a list of completions.

Parameters **text** : string

A string of text to be completed on. It can be given as empty and instead a line/position pair are given. In this case, the completer itself will split the line like readline does.

line [string, optional] The complete line that text is part of.

cursor_pos [int, optional] The position of the cursor on the input line.

Returns **text** : string

The actual text that was completed.

matches [list] A sorted list with all possible completions.

The optional arguments allow the completion to take more context into account, and are part of the low-level completion API. :

This is a wrapper around the completion mechanism, similar to what : readline does at the command line when the TAB key is hit. By : exposing it as a method, it can be used by other non-readline : environments (such as GUIs) for text completion. :

Simple usage example: :

```
In [1]: x = 'hello' :
```

```
In [2]: _ip.complete('x.l') :
```

```
Out[2]: ('x.l', ['x.ljust', 'x.lower', 'x.lstrip']) :
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

db

debug

A casting version of the boolean trait.

debugger (*force=False*)

Call the pydb/pdb debugger.

Keywords:

- force(False)**: by default, this routine checks the instance `call_pdb`

flag and does not actually invoke the debugger if the flag is false. The ‘force’ option forces the debugger to activate even if the flag is false.

deep_reload

A casting version of the boolean trait.

default_option (*fn, optstr*)

Make an entry in the `options_table` for `fn`, with value `optstr`

define_macro (*name, themacro*)

Define a new macro

Parameters **name** : str

The name of the macro.

themacro : str or Macro

The action to do upon invoking the macro. If a string, a new Macro object is created by passing the string to it.

define_magic (*magicname, func*)

Expose own function as magic function for ipython

```
def foo_impl(self,parameter_s=''): 'My very own magic!. (Use docstrings, IPython reads them).' print 'Magic function. Passed parameter is between < >:' print '<%s>' % parameter_s print 'The self object is:',self
```

```
self.define_magic('foo',foo_impl)
```

display_formatter

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

display_pub_class

A trait whose value must be a subclass of a specified class.

display_trap

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

displayhook_class

A trait whose value must be a subclass of a specified class.

enable_pylab (*gui=None*)

ev (*expr*)

Evaluate python expression *expr* in user namespace.

Returns the result of evaluation

ex (*cmd*)

Execute a normal python statement in user namespace.

excepthook (*etype, value, tb*)

One more defense for GUI apps that call `sys.excepthook`.

GUI frameworks like wxPython trap exceptions and call `sys.excepthook` themselves. I guess this is a feature that enables them to keep running after exceptions that would otherwise kill their mainloop. This is a bother for IPython which expects to catch all of the program exceptions with a `try: except: statement`.

Normally, IPython sets `sys.excepthook` to a `CrashHandler` instance, so if any app directly invokes `sys.excepthook`, it will look to the user like IPython crashed. In order to work around this, we can disable the `CrashHandler` and replace it with this `excepthook` instead, which prints a regular traceback using our `InteractiveTB`. In this fashion, apps which call `sys.excepthook` will generate a regular-looking exception from IPython, and the `CrashHandler` will only be triggered by real IPython crashes.

This hook should be used sparingly, only in places which are not likely to be true IPython errors.

execution_count

A integer trait.

exit_now

A casting version of the boolean trait.

extension_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

extract_input_slices (*slices, raw=False*)

Return as a string a set of input history slices.

Inputs:

- *slices*: the set of slices is given as a list of strings (like `['1', '4:8', '9']`, since this function is for use by magic functions which get their arguments as strings.

Optional inputs:

- *raw(False)*: by default, the processed input is used. If this is `true`, the raw input history is used instead.

Note that *slices* can be called with two notations:

`N:M` -> standard python form, means including items `N...(M-1)`.

`N-M` -> include items `N..M` (closed endpoint).

filename

A trait for strings.

format_latex (*strng*)

Format a string for latex inclusion.

get_history (*index=None, raw=False, output=True*)

get_ipython ()

Return the currently running IPython instance.

getoutput (*cmd, split=True*)

Get output (possibly including stderr) from a subprocess.

Parameters **cmd** : str

Command to execute (can not end in '&', as background processes are not supported).

split : bool, optional

If True, split the output into an IPython SList. Otherwise, an IPython LSString is returned. These are objects similar to normal lists and strings, with a few convenience attributes for easier manipulation of line-based output. You can use '?' on them for details.

history_length

A integer trait.

history_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

history_saving_wrapper (*func*)

Wrap func for readline history saving

Convert func into callable that saves & restores history around the call

init_alias ()

init_builtins ()

init_completer ()

Initialize the completion machinery.

This creates completion machinery that can be used by client code, either interactively in-process (typically triggered by the readline library), programatically (such as in test suites) or out-of-process (typically over the network by remote frontends).

init_create_namespaces (*user_ns=None, user_global_ns=None*)

init_display_formatter ()

init_display_pub ()

init_displayhook ()

init_encoding ()

init_environment()
Any changes we need to make to the user's environment.

init_extension_manager()

init_history()
Sets up the command history, and starts regular autosaves.

init_hooks()

init_inspector()

init_instance_attrs()

init_io()

init_ipython_dir(ipython_dir)

init_logger()

init_logstart()
Initialize logging in case it was requested at the command line.

init_magics()

init_payload()

init_pdb()

init_plugin_manager()

init_prefilter()

init_prompts()

init_pushd_popd_magic()

init_readline()
Command history completion/saving/reloading.

init_reload_doctest()

init_syntax_highlighting()

init_sys_modules()

init_traceback_handlers(custom_exceptions)

init_user_ns()
Initialize all user-visible namespaces to their minimum defaults.
Certain history lists are also initialized here, as they effectively act as user namespaces.

Notes

All data structures here are only filled in, they are NOT reset by this method. If they were not empty before, data will simply be added to them.

classmethod initialized()

input_splitter

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

classmethod instance (*args, **kwargs)

Returns a global InteractiveShell instance.

ipython_dir

A trait for unicode strings.

logappend

A trait for strings.

logfile

A trait for strings.

logstart

A casting version of the boolean trait.

lsmagic ()

Return a list of currently available magic functions.

Gives a list of the bare names after mangling (['ls', 'cd', ...], not ['magic_ls', 'magic_cd', ...])

magic (arg_s)

Call a magic function by name.

Input: a string containing the name of the magic function to call and any additional arguments to be passed to the magic.

magic('name -opt foo bar') is equivalent to typing at the ipython prompt:

```
In[1]: %name -opt foo bar
```

To call a magic without arguments, simply use magic('name').

This provides a proper Python function to call IPython's magics in any valid Python code you can type at the interpreter, including loops and compound statements.

magic_Exit (parameter_s='')

Exit IPython.

magic_Quit (parameter_s='')

Exit IPython.

magic_alias (parameter_s='')

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 `$PATH`. See the `%rehash` and `%rehashx` functions, which automatically create aliases for the contents of your `$PATH`.

If called with no parameters, `%alias` prints the current alias table.

`magic_autocall` (*parameter_s*='')

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 (remembering the previous state).

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)

magic_automagic (*parameter_s=''*)

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

Without arguments it 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.

magic_bookmark (*parameter_s=''*)

Manage IPython's bookmark system.

%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.

magic_cd (*parameter_s=''*)

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'`.

`magic_colors` (*parameter_s=''*)

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

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

`magic_debug` (*parameter_s=''*)

Activate the interactive debugger in post-mortem mode.

If an exception has just occurred, this lets you inspect its stack frames interactively. Note that this will always work only on the last traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because if another one occurs, it clobbers the previous one.

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

`magic_dhist` (*parameter_s=''*)

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` (`n1` 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>`.

`magic_dirs` (*parameter_s=''*)

Return the current directory stack.

`magic_doctest_mode` (*parameter_s=''*)

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.

magic_ed (*parameter_s*='')

Alias to %edit.

magic_edit (*parameter_s*='', *last_call*=['', ''])

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 `__IPYTHON__.rc.editor` command. This is read from your environment variable \$EDITOR. 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 command line option '-editor' or in your ipythonrc file. This is useful if you wish to use specifically for IPython an editor different from your typical default (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:

- The arguments are numbers or pairs of colon-separated numbers (like 1 4:8 9). These are interpreted as lines of previous input to be loaded into the editor. The syntax is the same of the %macro command.

- If the argument doesn't start with a number, it is evaluated as a variable and its contents 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.

- If the argument is not found as a variable, IPython will look for a

file with that name (adding `.py` if necessary) and load it into the editor. It will execute its contents with `execfile()` when you exit, loading any code in the file into your interactive namespace.

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]: ed Editing... done. Executing edited code... Out[1]: 'def foo():\n print "foo() was defined\n 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]: ed 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]: ed Editing... done. Executing edited code... hello Out[5]: "print 'hello'\n"
```

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

```
In [6]: ed _ 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]: ed _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.

`magic_env` (*parameter_s*='')

List environment variables.

`magic_exit` (*parameter_s*='')

Exit IPython.

`magic_gui` (*parameter_s*='')

Enable or disable IPython GUI event loop integration.

`%gui [GUINAME]`

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

```
%gui wx      # enable wxPython event loop integration
%gui qt4|qt   # enable PyQt4 event loop integration
%gui gtk      # enable PyGTK event loop integration
%gui tk       # enable Tk event loop integration
%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.

`magic_install_default_config` (*s*)

Install IPython's default config file into the `.ipython` dir.

If the default config file (`ipython_config.py`) is already installed, it will not be overwritten. You can force overwriting by using the `-o` option:

```
In [1]: %install_default_config
```

`magic_install_profiles` (*s*)

Install the default IPython profiles into the `.ipython` dir.

If the default profiles have already been installed, they will not be overwritten. You can force overwriting them by using the `-o` option:

```
In [1]: %install_profiles -o
```

`magic_load_ext` (*module_str*)

Load an IPython extension by its module name.

`magic_logoff` (*parameter_s*='')

Temporarily stop logging.

You must have previously started logging.

`magic_logon` (*parameter_s*='')

Restart logging.

This function is for restarting logging which you've temporarily stopped with `%logoff`. For starting logging for the first time, you must use the `%logstart` function, which allows you to specify an optional log filename.

magic_logstart (*parameter_s*='')

Start logging anywhere in a session.

`%logstart [-ol-rl-t] [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: well, that says it.backup: rename (if exists) to name~ and start name.global: single logfile in your home dir, appended to.over : overwrite existing log.rotate: create rotating logs name.1~, name.2~, etc.

Options:

-o: log also IPython's output. In this mode, all commands which generate an Out[NN] prompt are recorded to the logfile, right after their corresponding input line. The output lines are always prepended with a '#[Out]# ' marker, so that the log remains valid Python code.

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 in their final form, converted into valid Python. For example, `%Exit` is logged as `'_ip.magic("Exit")`. If the -r flag is given, all input is logged exactly as typed, with no transformations applied.

-t: put timestamps before each input line logged (these are put in comments).

magic_logstate (*parameter_s*='')

Print the status of the logging system.

magic_logstop (*parameter_s*='')

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.

magic_lsmagic (*parameter_s*='')

List currently available magic functions.

magic_macro (*parameter_s*='')

Define a set of input lines as a macro for future re-execution.

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 as the command line is used instead.

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 notation for indicating number ranges is: *n1-n2* means 'use line numbers *n1,...n2*' (the endpoint is included). That is, '5-7' means using the lines numbered 5,6 and 7.

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 (%hist prints it):

```
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'.
```

For one-off cases which DON'T contain magic function calls in them you can obtain similar results by explicitly executing slices from your input history with:

```
In [60]: exec In[44:48]+In[49]
```

magic_magic (*parameter_s*='')

Print information about the magic function system.

Supported formats: -latex, -brief, -rest

magic_page (*parameter_s*='')

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.

magic_pdb (*parameter_s=''*)

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 ipythonrc configuration file (the variable is called '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.

magic_pdef (*parameter_s='', namespaces=None*)

Print the definition header for any callable object.

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

magic_pdoc (*parameter_s='', namespaces=None*)

Print the docstring for an object.

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

magic_pfile (*parameter_s=''*)

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.

magic_pinfo (*parameter_s='', namespaces=None*)

Provide detailed information about an object.

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

magic_pinfo2 (*parameter_s='', namespaces=None*)

Provide extra detailed information about an object.

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

magic_popd (*parameter_s=''*)

Change to directory popped off the top of the stack.

magic_pprint (*parameter_s=''*)

Toggle pretty printing on/off.

magic_profile (*parameter_s=''*)

Print your currently active IPython profile.

magic_prun (*parameter_s='', user_mode=1, opts=None, arg_lst=None, prog_ns=None*)

Run a statement through the python code profiler.

Usage: `%prun [options] statement`

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 namespaces which do not hold under IPython.

Options:

`-l <limit>`: you can place restrictions on what or how much of the profile gets printed. The limit value can be:

- 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, use a limit of 0.4 to see the topmost 40% only).

You can combine several limits with repeated use of the option. For example, `'-l __init__ -l 5'` will print only the topmost 5 lines of information about class constructors.

`-r`: return the `pstats.Stats` object generated by the profiling. This object has all the information about the profile in it, and you can later use it for further analysis or in other functions.

`-s <key>`: sort profile by given key. You can provide more than one key by using the option several times: `'-s key1 -s key2 -s key3...'`. The default sorting key is 'time'.

The following is copied verbatim from the profile documentation referenced below:

When more than one key is provided, additional keys are used as secondary criteria when there is equality in all keys selected before them.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

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

Note that all sorts on statistics are in descending order (placing most time consuming items first), whereas name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between "nfl" and "stdname" is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order "20" "3" and "40". In contrast, "nfl" does a numeric compare of the line numbers. In fact, `sort_stats("nfl")` is the same as `sort_stats("name", "file", "line")`.

`-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 understood by the `pstats` module, and is generated by a call to the `dump_stats()` method of profile objects. The profile is still shown on screen.

If you want to run complete programs under the profiler's control, use `'%run -p [prof_opts] filename.py [args to program]'` where `prof_opts` contains profiler specific options as described here.

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

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

magic_psearch (*parameter_s=''*)

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:

-a: makes the pattern match even objects whose names start with a single underscore. These names are normally omitted from the search.

-i/-c: make the pattern case insensitive/sensitive. If neither of these options is given, the default is read from your `ipythonrc` file. The option name which sets this value is `'wildcards_case_sensitive'`. If this option is not specified in your `ipythonrc` 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

`magic_psource` (*parameter_s='', namespaces=None*)

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

`magic_pushd` (*parameter_s=''*)

Place the current dir on stack and change directory.

Usage: `%pushd` [`'dirname'`]

`magic_pwd` (*parameter_s=''*)

Return the current working directory path.

`magic_pycat` (*parameter_s=''*)

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.

`magic_pylab` (*s*)

Load numpy and matplotlib to work interactively.

`%pylab` [`GUINAME`]

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

It will import at the top level numpy as np, pyplot as plt, matplotlib, pylab and mlab, as well as all names from numpy and pylab.

Parameters `guiname` : optional

One of the valid arguments to the `%gui` magic (`'qt'`, `'wx'`, `'gtk'` or `'tk'`). If given, the corresponding Matplotlib backend is used, otherwise matplotlib's default (which you can override in your matplotlib config file) is used.

Examples

In this case, where the MPL default is TkAgg: In [2]: `%pylab`

Welcome to pylab, a matplotlib-based Python environment. Backend in use: TkAgg For more information, type 'help(pylab)'.

But you can explicitly request a different backend: In [3]: `%pylab qt`

Welcome to pylab, a matplotlib-based Python environment. Backend in use: Qt4Agg For more information, type 'help(pylab)'.

`magic_quickref` (*arg*)

Show a quick reference sheet

`magic_quit` (*parameter_s*='')

Exit IPython.

`magic_r` (*parameter_s*='')

Repeat previous input.

Note: Consider using the more powerfull `%rep` instead!

If given an argument, repeats the previous command which starts with the same string, otherwise it just repeats the previous input.

Shell escaped commands (with ! as first character) are not recognized by this system, only pure python code and magic commands.

`magic_rehashx` (*parameter_s*='')

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

This version explicitly checks that every entry in \$PATH is a file with execute access (os.X_OK), so it is much slower than `%rehash`.

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

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

`magic_reload_ext` (*module_str*)

Reload an IPython extension by its module name.

`magic_reset` (*parameter_s*='')

Resets the namespace by removing all names defined by the user.

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

Parameters `-y` : force reset without asking for confirmation.

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 [10]: 'a' in _ip.user_ns Out[10]: False
```

magic_reset_selective (*parameter_s=''*)

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.

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']
```

magic_run (*parameter_s=''*, *runner=None*, *file_finder=<function get_py_filename at 0x104ab4c08>*)

Run the named file inside IPython as a program.

Usage: `%run [-n -i -t [-N<N>] -d [-b<N>] -p [profile options]] 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 use (unless `-p` is used, see below).

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.

Options:

`-n`: `__name__` is NOT set to `'__main__'`, but to the running file's name without extension (as python does under `import`). This allows running scripts and reloading the definitions in them without calling code protected by an `'if __name__ == "__main__":'` clause.

`-i`: run the file in IPython's namespace instead of an empty one. This is useful if you are experimenting with code written in a text editor which depends on variables defined interactively.

`-e`: ignore `sys.exit()` calls or `SystemExit` exceptions in the script being run. This is particularly useful if IPython is being used to run unittests, which always exit with a `sys.exit()` call. In such cases you are interested in the output of the test results, not in seeing a traceback of the unittest module.

`-t`: print timing information at the end of the run. IPython will give you an estimated CPU time consumption for your script, which under Unix uses the `resource` module to avoid the wraparound problems of `time.clock()`. Under Unix, an estimate of time spent on system tasks is also given (for Windows platforms this is reported as 0.0).

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 runUser : 0.910862 s, 0.1821724 s.System: 0.0 s, 0.0 s.
```

`-d`: run your program under the control of `pdb`, the Python debugger. This allows you to execute your program step by step, watch variables, etc. Internally, what IPython does is similar to calling:

```
pdb.run('execfile("YOURFILENAME")')
```

with a breakpoint set on line 1 of your file. You can change the line number for this automatic breakpoint to be `<N>` by using the `-bN` option (where `N` must be an integer). For example:

```
%run -d -b40 myscript
```

will set the first breakpoint at line 40 in `myscript.py`. Note that the first breakpoint must be set on a line which actually does something (not a comment or docstring) for it to stop execution.

When the `pdb` debugger starts, you will see a `(Pdb)` prompt. You must first enter `'c'` (without quotes) to start execution up to the first breakpoint.

Entering `'help'` gives information about the use of the debugger. You can easily see `pdb`'s full documentation with `"import pdb;pdb.help()"` at a prompt.

`-p`: run program under the control of the Python profiler module (which prints a detailed report of execution times, function calls, etc).

You can pass other options after `-p` which affect the behavior of the profiler itself. See the docs for `%prun` for details.

In this mode, the program's variables do NOT propagate back to the IPython interactive namespace (because they remain in the namespace where the profiler executes them).

Internally this triggers a call to `%prun`, see its documentation for details on the options available specifically for profiling.

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

`magic_save` (*parameter_s=''*)

Save a set of lines 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.

This function uses the same syntax as `%macro` for line extraction, but instead of creating a macro it saves the resulting string 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.

`magic_sc` (*parameter_s=''*)

Shell capture - execute a shell command and capture its output.

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 space-separated string. These are convenient, respectively, either for sequential processing or to be passed to a shell command.

For example:

all-random

```
# Capture into variable a In [1]: sc a=ls *py
# a is a string with embedded newlines In [2]: a Out[2]:
'setup.pynwin32_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.

magic_sx (*parameter_s*='')

Shell execute - run a shell command and capture its output.

%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.

magic_tb (*s*)

Print the last traceback with the currently active exception mode.

See %xmode for changing exception reporting modes.

magic_time (*parameter_s*='')

Time execution of a Python statement or expression.

The CPU and wall clock times are printed, and the value of the expression (if any) is returned. Note that under Win32, system time is always reported as 0, since it can not be measured.

This function provides very basic timing functionality. In Python 2.3, the `timeit` module offers more control and sophistication, so this could be rewritten to use it (patches welcome).

Some 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

magic_timeit (*parameter_s*='')

Time execution of a Python statement or expression

Usage: %timeit [-n<N> -r<R> [-tl-c]] statement

Time execution of a Python statement or expression using the timeit module.

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

Examples:

In [1]: %timeit pass 10000000 loops, best of 3: 53.3 ns per loop

In [2]: u = None

In [3]: %timeit u is None 10000000 loops, best of 3: 184 ns per loop

In [4]: %timeit -r 4 u == None 1000000 loops, best of 4: 242 ns per loop

In [5]: import time

In [6]: %timeit -n1 time.sleep(2) 1 loops, best of 3: 2 s per loop

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.

magic_unalias (*parameter_s*='')

Remove an alias

magic_unload_ext (*module_str*)

Unload an IPython extension by its module name.

magic_who (*parameter_s*='')

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.

magic_who_ls (*parameter_s*='')

Return a sorted list of all interactive variables.

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

magic_whos (*parameter_s*='')

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 and Numeric arrays, a summary with shape, number of elements, typecode and size in memory.
- Everything else: a string representation, snipping their middle if too long.

magic_xmode (*parameter_s*='')

Switch modes for the exception handlers.

Valid modes: Plain, Context and Verbose.

If called without arguments, acts as a toggle.

make_user_namespaces (*user_ns=None, user_global_ns=None*)

Return a valid local and global user interactive namespaces.

This builds a dict with the minimal information needed to operate as a valid IPython user namespace, which you can pass to the various embedding classes in ipython. The default implementation returns the same dict for both the locals and the globals to allow functions to refer to variables in the namespace. Customized implementations can return different dicts. The locals dictionary can actually be anything following the basic mapping protocol of a dict, but the globals dict must be a true dict, not even a subclass. It is recommended that any custom object for the locals namespace synchronize with the globals dict somehow.

Raises `TypeError` if the provided globals namespace is not a true dict.

Parameters `user_ns` : dict-like, optional

The current user namespace. The items in this namespace should be included in the output. If `None`, an appropriate blank namespace should be created.

`user_global_ns` : dict, optional

The current user global namespace. The items in this namespace should be included in the output. If `None`, an appropriate blank namespace should be created.

Returns A pair of dictionary-like object to be used as the local namespace :

of the interpreter and a dict to be used as the global namespace.

mktempfile (*data=None, prefix='ipython_edit_'*)

Make a new tempfile and return its filename.

This makes a call to `tempfile.mktemp`, but it registers the created filename internally so ipython cleans it up at exit time.

Optional inputs:

- `data(None)`: if data is given, it gets written out to the temp file immediately, and the file is closed again.

new_main_mod (*ns=None*)

Return a new 'main' module object for user code execution.

object_info_string_level

An enum that whose value must be in a given sequence.

object_inspect (*oname*)

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

parse_options (*arg_str, opt_str, *long_opts, **kw*)

Parse options passed to an argument string.

The interface is similar to that of `getopt()`, but it returns back a Struct with the options as keys and the stripped argument string still as a string.

`arg_str` is quoted as a true `sys.argv` vector by using `shlex.split`. This allows us to easily expand variables, glob files, quote arguments, etc.

Options: `-mode`: default 'string'. If given as 'list', the argument string is returned as a list (split on whitespace) instead of a string.

`-list_all`: put all option values in lists. Normally only options appearing more than once are put in a list.

`-posix` (True): whether to split the input line in POSIX mode or not, as per the conventions outlined in the `shlex` module from the standard library.

payload_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

pdb

A casting version of the boolean trait.

plugin_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

pre_readline ()

readline hook to be used at the start of each line.

Currently it handles auto-indent only.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

profile

A trait for strings.

profile_missing_notice (**args, **kwargs*)

prompt_in1

A trait for strings.

prompt_in2

A trait for strings.

prompt_out

A trait for strings.

prompts_pad_left

A casting version of the boolean trait.

push (*variables*, *interactive=True*)

Inject a group of variables into the IPython user namespace.

Parameters **variables** : dict, str or list/tuple of str

The variables to inject into the user's namespace. If a dict, a simple update is done. If a str, the string is assumed to have variable names separated by spaces. A list/tuple of str can also be used to give the variable names. If just the variable names are give (list/tuple/str) then the variable values looked up in the callers frame.

interactive : bool

If True (default), the variables will be listed with the `who` magic.

push_line (*line*)

Push a line to the interpreter.

The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `run_source()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is 1 if more input is required, 0 if the line was dealt with in some way (this is the same as `run_source()`).

quiet

A casting version of the boolean trait.

readline_merge_completions

A casting version of the boolean trait.

readline_omit__names

An enum that whose value must be in a given sequence.

readline_parse_and_bind

An instance of a Python list.

readline_remove_delims

A trait for strings.

readline_use

A casting version of the boolean trait.

register_post_execute (*func*)

Register a function for calling after code execution.

reload_history()

Reload the input history from disk file.

reset()

Clear all internal namespaces.

Note that this is much more aggressive than %reset, since it clears fully all namespaces, as well as all input/output lists.

reset_buffer()

Reset the input buffer.

reset_selective (*regex=None*)

Clear selective variables from internal namespaces based on a specified regular expression.

Parameters **regex** : string or compiled pattern, optional

A regular expression pattern that will be used in searching variable names in the users namespaces.

resetbuffer()

Reset the input buffer.

restore_sys_module_state()

Restore the state of the sys module.

run_cell (*cell*)

Run the contents of an entire multiline ‘cell’ of code.

The cell is split into separate blocks which can be executed individually. Then, based on how many blocks there are, they are executed as follows:

- A single block: ‘single’ mode.

If there’s more than one block, it depends:

- if the last one is no more than two lines long, run all but the last

in ‘exec’ mode and the very last one in ‘single’ mode. This makes it easy to type simple expressions at the end to see computed values. - otherwise (last one is also multiline), run all in ‘exec’ mode

When code is executed in ‘single’ mode, `sys.displayhook()` fires, results are displayed and output prompts are computed. In ‘exec’ mode, no results are displayed unless `print()` is called explicitly; this mode is more akin to running a script.

Parameters **cell** : str

A single or multiline string.

run_code (*code_obj*, *post_execute=True*)

Execute a code object.

When an exception occurs, `self.showtraceback()` is called to display a traceback.

Return value: a flag indicating whether the code to be run completed successfully:

- 0: successful execution.

- 1: an error occurred.

run_one_block (*block*)

Run a single interactive block of source code.

If the block is single-line, dynamic transformations are applied to it (like automagics, autocall and alias recognition).

If the block is multi-line, it must consist of valid Python code only.

Parameters **block** : string

A (possibly multiline) string of code to be executed.

Returns The output of the underlying execution method used, be it :

:meth:'run_source' or :meth:'run_single_line'. :

run_single_line (*line*)

Run a single-line interactive statement.

This assumes the input has been transformed to IPython syntax by applying all static transformations (those with an explicit prefix like % or !), but it will further try to apply the dynamic ones.

It does not update history.

run_source (*source, filename=None, symbol='single', post_execute=True*)

Compile and run some source in the interpreter.

Arguments are as for compile_command().

One several things can happen:

- 1) The input is incorrect; compile_command() raised an exception (SyntaxError or OverflowError). A syntax traceback will be printed by calling the showsyntaxerror() method.
- 2) The input is incomplete, and more input is required; compile_command() returned None. Nothing happens.
- 3) The input is complete; compile_command() returned a code object. The code is executed by calling self.run_code() (which also handles run-time exceptions, except for SystemExit).

The return value is:

- True in case 2
- False in the other cases, unless an exception is raised, where

None is returned instead. This can be used by external callers to know whether to continue feeding input or not.

The return value can be used to decide whether to use sys.ps1 or sys.ps2 to prompt the next line.

run_code (*code_obj, post_execute=True*)

Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Return value: a flag indicating whether the code to be run completed successfully:

- 0: successful execution.
- 1: an error occurred.

runlines (*lines*, *clean=False*)

Run a string of one or more lines of source.

This method is capable of running a string containing multiple source lines, as if they had been entered at the IPython prompt. Since it exposes IPython's processing machinery, the given strings can contain magic calls (%magic), special shell access (!cmd), etc.

runsource (*source*, *filename=None*, *symbol='single'*, *post_execute=True*)

Compile and run some source in the interpreter.

Arguments are as for `compile_command()`.

One several things can happen:

- 1) The input is incorrect; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`). A syntax traceback will be printed by calling the `showsyntaxerror()` method.
- 2) The input is incomplete, and more input is required; `compile_command()` returned `None`. Nothing happens.
- 3) The input is complete; `compile_command()` returned a code object. The code is executed by calling `self.run_code()` (which also handles run-time exceptions, except for `SystemExit`).

The return value is:

- True in case 2
- False in the other cases, unless an exception is raised, where

None is returned instead. This can be used by external callers to know whether to continue feeding input or not.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

safe_execfile (*fname*, **where*, ***kw*)

A safe version of the builtin `execfile()`.

This version will never throw an exception, but instead print helpful error messages to the screen. This only works on pure Python files with the `.py` extension.

Parameters **fname** : string

The name of the file to be executed.

where : tuple

One or two namespaces, passed to `execfile()` as (`globals`, `locals`). If only one is given, it is passed as both.

exit_ignore : bool (False)

If True, then silence `SystemExit` for non-zero status (it is always silenced for zero status, as it is so common).

safe_execfile_ipy (*fname*)

Like `safe_execfile`, but for `.ipy` files with IPython syntax.

Parameters `fname` : str

The name of the file to execute. The filename must have a `.ipy` extension.

save_history ()

Save input history to a file (via readline library).

save_sys_module_state ()

Save the state of hooks in the `sys` module.

This has to be called after `self.user_ns` is created.

separate_in

A Str subclass to validate `separate_in`, `separate_out`, etc.

This is a Str based trait that converts `'0'->` and `'n'->`

‘.

separate_out

A Str subclass to validate `separate_in`, `separate_out`, etc.

This is a Str based trait that converts `'0'->` and `'n'->`

‘.

separate_out2

A Str subclass to validate `separate_in`, `separate_out`, etc.

This is a Str based trait that converts `'0'->` and `'n'->`

‘.

set_autoindent (*value=None*)

Set the autoindent flag, checking for readline support.

If called with no arguments, it acts as a toggle.

set_completer_frame (*frame=None*)

Set the frame of the completer.

set_custom_completer (*completer, pos=0*)

Adds a new custom completer function.

The position argument (defaults to 0) is the index in the completers list where you want the completer to be inserted.

set_custom_exc (*exc_tuple, handler*)

Set a custom exception handler, which will be called if any of the exceptions in `exc_tuple` occur in the mainloop (specifically, in the `run_code()` method).

Inputs:

- `exc_tuple`: a *tuple* of valid exceptions to call the defined

handler for. It is very important that you use a tuple, and NOT A LIST here, because of the way Python's except statement works. If you only want to trap a single exception, use a singleton tuple:

```
exc_tuple == (MyCustomException,)
```

- handler: this must be defined as a function with the following

basic interface:

```
def my_handler(self, etype, value, tb, tb_offset=None)
    ...
    # The return value must be
    return structured_traceback
```

This will be made into an instance method (via `types.MethodType`) of IPython itself, and it will be called if any of the exceptions listed in the `exc_tuple` are caught. If the handler is `None`, an internal basic one is used, which just prints basic info.

WARNING: by putting in your own exception handler into IPython's main execution loop, you run a very good chance of nasty crashes. This facility should only be used if you really know what you are doing.

set_hook (*name, hook, priority=50, str_key=None, re_key=None*)
set_hook(name, hook) -> sets an internal IPython hook.

IPython exposes some of its internal API as user-modifiable hooks. By adding your function to one of these hooks, you can modify IPython's behavior to call at runtime your own routines.

set_next_input (*s*)
Sets the 'default' input string for the next command line.
Requires readline.

Example:

```
[D:ipython]|1> _ip.set_next_input("Hello Word") [D:ipython]|2> Hello Word # cursor is here
```

set_readline_completer ()
Reset readline's completer to be our own.

show_usage ()
Show a usage message

showsyntaxerror (*filename=None*)
Display the syntax error that just occurred.

This doesn't display a stack trace because there isn't one.

If a filename is given, it is stuffed in the exception instead of what was there before (because Python's parser always uses "<string>" when reading from a string).

showtraceback (*exc_tuple=None, filename=None, tb_offset=None, exception_only=False*)
Display the exception that just occurred.

If nothing is known about the exception, this is the method which should be used throughout the code for presenting user tracebacks, rather than directly invoking the InteractiveTB object.

A specific `showsyntaxerror()` also exists, but this method can take care of calling it if needed, so unless you are explicitly catching a `SyntaxError` exception, don't try to analyze the stack manually and simply call this method.

system (*cmd*)

Call the given *cmd* in a subprocess.

Parameters *cmd* : str

Command to execute (can not end in '&', as background processes are not supported).

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

user_expressions (*expressions*)

Evaluate a dict of expressions in the user's namespace.

Parameters *expressions* : dict

A dict with string keys and string values. The expression values should be valid Python expressions, each of which will be evaluated in the user namespace.

Returns A dict, keyed like the input expressions dict, with the `repr()` of each :

value. :

user_variables (*names*)

Get a list of variable names from the user's namespace.

Parameters *names* : list of strings

A list of names of variables to be read from the user namespace.

Returns A dict, keyed by the input names and with the `repr()` of each value. :

var_expand (*cmd*, *depth=0*)

Expand python variables in a string.

The depth argument indicates how many frames above the caller should be walked to look for the local namespace where to expand variables.

The global namespace for expansion is always the user's interactive namespace.

wildcards_case_sensitive

A casting version of the boolean trait.

write (*data*)

Write a string to the default output

write_err (*data*)

Write a string to the default error output

xmode

An enum of strings that are caseless in validate.

InteractiveShellABC

class IPython.core.interactiveshell.InteractiveShellABC

Bases: object

An abstract base class for InteractiveShell.

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

MultipleInstanceError

class IPython.core.interactiveshell.MultipleInstanceError

Bases: exceptions.Exception

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

SeparateStr

class IPython.core.interactiveshell.SeparateStr (*default_value=<IPython.utils.traitlets.NoDefaultSpecified object at 0x104af1650>, **metadata*)

Bases: IPython.utils.traitlets.Str

A Str subclass to validate separate_in, separate_out, etc.

This is a Str based trait that converts '0'->'>' and 'n'->'

‘.

__init__ (*default_value=<IPython.utils.traitlets.NoDefaultSpecified object at 0x104af1650>, **metadata*)

Create a TraitType.

error (*obj*, *value*)

get_default_value ()

Create a new instance of the default value.

get_metadata (*key*)

info ()

init ()

instance_init (*obj*)

This is called by `HasTraits.__new__()` to finish init'ing.

Some stages of initialization must be delayed until the parent `HasTraits` instance has been created. This method is called in `HasTraits.__new__()` after the instance has been created.

This method trigger the creation and validation of default values and also things like the resolution of str given class names in `Type` and `:class'Instance'`.

Parameters **obj**: `HasTraits` instance

The parent `HasTraits` instance that has just been created.

set_default_value (*obj*)

Set the default value on a per instance basis.

This method is called by `instance_init()` to create and validate the default value. The creation and validation of default values must be delayed until the parent `HasTraits` class has been instantiated.

set_dynamic_initializer (*method*)

Set the dynamic initializer method, if any.

set_metadata (*key*, *value*)

validate (*obj*, *value*)

SpaceInInput

class `IPython.core.interactiveshell.SpaceInInput`

Bases: `exceptions.Exception`

__init__ ()

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

args

message

8.22.3 Functions

`IPython.core.interactiveshell.get_default_colors()`

```
IPython.core.interactiveshell.no_op(*a, **kw)
IPython.core.interactiveshell.softspace(file, newvalue)
    Copied from code.py, to remove the dependency
```

8.23 core.ipapi

8.23.1 Module: core.ipapi

This module is *completely* deprecated and should no longer be used for any purpose. Currently, we have a few parts of the core that have not been componentized and thus, still rely on this module. When everything has been made into a component, this module will be sent to deathrow.

```
IPython.core.ipapi.get()
    Get the global InteractiveShell instance.
```

8.24 core.logger

8.24.1 Module: core.logger

Inheritance diagram for IPython.core.logger:



```
graph TD
    A[core.logger.Logger]
```

Logger class for IPython's logging facilities.

8.24.2 Logger

```
class IPython.core.logger.Logger(home_dir, logfname='Logger.log', loghead='', log-
                                mode='over')
```

Bases: object

A Logfile class with different policies for file creation

```
__init__(home_dir, logfname='Logger.log', loghead='', logmode='over')
```

```
close_log()
```

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.

log (*line_mod*, *line_ori*)

Write the sources to a log.

Inputs:

- line_mod*: possibly modified input, such as the transformations made by input prefilters or input handlers of various kinds. This should always be valid Python.
- line_ori*: unmodified input line from the user. This is not necessarily valid Python.

log_write (*data*, *kind*='input')

Write data to the log file, if active

logmode

logstart (*logfname*=None, *loghead*=None, *logmode*=None, *log_output*=False, *timestamp*=False, *log_raw_input*=False)

Generate a new log-file with a default header.

Raises RuntimeError if the log has already been started

logstate ()

Print a status message about the logger.

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.

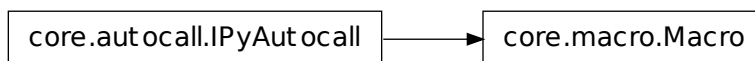
switch_log (*val*)

Switch logging on/off. *val* should be ONLY a boolean.

8.25 core.macro

8.25.1 Module: core.macro

Inheritance diagram for IPython.core.macro:



Support for interactive macros in IPython

8.25.2 Macro

class `IPython.core.magic.Macro` (*data*)

Bases: `IPython.core.autocall.IPyAutocall`

Simple class to store the value of macros as strings.

Macro is just a callable that executes a string of IPython input when called.

Args to macro are available in `_margv` list if you need them.

__init__ (*data*)

store the macro value, as a single string which can be executed

set_ip (*ip*)

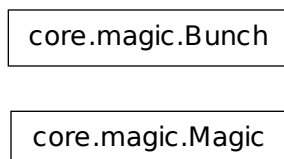
Will be used to set `_ip` point to current ipython instance b/f call

Override this method if you don't want this to happen.

8.26 core.magic

8.26.1 Module: `core.magic`

Inheritance diagram for `IPython.core.magic`:



Magic functions for InteractiveShell.

8.26.2 Classes

Bunch

class `IPython.core.magic.Bunch`

Magic

class `IPython.core.magic.Magic` (*shell*)

Magic functions for InteractiveShell.

Shell functions which can be reached as `%function_name`. All magic functions should accept a string, which they can parse for their own needs. This can make some functions easier to type, eg `%cd ../` vs. `%cd("../")`

ALL definitions MUST begin with the prefix `magic`. The user won't need it at the command line, but it is needed in the definition.

__init__ (*shell*)

arg_err (*func*)

Print docstring if incorrect arguments were passed

default_option (*fn, optstr*)

Make an entry in the `options_table` for `fn`, with value `optstr`

extract_input_slices (*slices, raw=False*)

Return as a string a set of input history slices.

Inputs:

- `slices`: the set of slices is given as a list of strings (like `['1', '4:8', '9']`, since this function is for use by magic functions which get their arguments as strings.

Optional inputs:

- `raw(False)`: by default, the processed input is used. If this is true, the raw input history is used instead.

Note that slices can be called with two notations:

`N:M` -> standard python form, means including items `N...(M-1)`.

`N-M` -> include items `N..M` (closed endpoint).

format_latex (*strng*)

Format a string for latex inclusion.

lsmagic ()

Return a list of currently available magic functions.

Gives a list of the bare names after mangling (`['ls', 'cd', ...]`, not `['magic_ls', 'magic_cd', ...]`)

magic_Exit (*parameter_s=''*)

Exit IPython.

magic_Quit (*parameter_s=''*)

Exit IPython.

magic_alias (*parameter_s=''*)

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 `$PATH`. See the `%rehash` and `%rehashx` functions, which automatically create aliases for the contents of your `$PATH`.

If called with no parameters, `%alias` prints the current alias table.

magic_autocall (*parameter_s*='')

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 (remembering the previous state).

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)

magic_automagic (*parameter_s=''*)

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

Without arguments it 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.

magic_bookmark (*parameter_s=''*)

Manage IPython's bookmark system.

`%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.

magic_cd (*parameter_s=''*)

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'`.

magic_colors (*parameter_s*='')

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

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

magic_debug (*parameter_s*='')

Activate the interactive debugger in post-mortem mode.

If an exception has just occurred, this lets you inspect its stack frames interactively. Note that this will always work only on the last traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because if another one occurs, it clobbers the previous one.

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

magic_dhist (*parameter_s*='')

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` (`n1` 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>`.

magic_dirs (*parameter_s*='')

Return the current directory stack.

magic_doctest_mode (*parameter_s*='')

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.

magic_ed (*parameter_s*='')
Alias to %edit.

magic_edit (*parameter_s*='', *last_call*=['', ''])
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 `__IPYTHON__.rc.editor` command. This is read from your environment variable `$EDITOR`. 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 command line option `'-editor'` or in your `ipythonrc` file. This is useful if you wish to use specifically for IPython an editor different from your typical default (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:

- The arguments are numbers or pairs of colon-separated numbers (like 1 4:8 9). These are interpreted as lines of previous input to be loaded into the editor. The syntax is the same of the %macro command.
- If the argument doesn't start with a number, it is evaluated as a

variable and its contents 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.

- If the argument is not found as a variable, IPython will look for a

file with that name (adding .py if necessary) and load it into the editor. It will execute its contents with `execfile()` when you exit, loading any code in the file into your interactive namespace.

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]: ed 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]: ed 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]: ed Editing... done. Executing edited code... hello Out[5]: “print ‘hello’n”
```

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

```
In [6]: ed _ 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]: ed _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.

`magic_env` (*parameter_s*='')

List environment variables.

`magic_exit` (*parameter_s*='')

Exit IPython.

`magic_gui` (*parameter_s*='')

Enable or disable IPython GUI event loop integration.

`%gui` [GUINAME]

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

```
%gui wx      # enable wxPython event loop integration
%gui qt4|qt   # enable PyQt4 event loop integration
%gui gtk      # enable PyGTK event loop integration
%gui tk       # enable Tk event loop integration
%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.

`magic_install_default_config` (*s*)

Install IPython's default config file into the `.ipython` dir.

If the default config file (`ipython_config.py`) is already installed, it will not be overwritten. You can force overwriting by using the `-o` option:

```
In [1]: %install_default_config
```

`magic_install_profiles` (*s*)

Install the default IPython profiles into the `.ipython` dir.

If the default profiles have already been installed, they will not be overwritten. You can force overwriting them by using the `-o` option:

```
In [1]: %install_profiles -o
```

`magic_load_ext` (*module_str*)

Load an IPython extension by its module name.

`magic_logoff` (*parameter_s*='')

Temporarily stop logging.

You must have previously started logging.

magic_logon (*parameter_s*='')

Restart logging.

This function is for restarting logging which you've temporarily stopped with %logoff. For starting logging for the first time, you must use the %logstart function, which allows you to specify an optional log filename.

magic_logstart (*parameter_s*='')

Start logging anywhere in a session.

%logstart [-ol-rl-t] [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: well, that says it.backup: rename (if exists) to name~ and start name.global: single logfile in your home dir, appended to.over : overwrite existing log.rotate: create rotating logs name.1~, name.2~, etc.

Options:

-o: log also IPython's output. In this mode, all commands which generate an Out[NN] prompt are recorded to the logfile, right after their corresponding input line. The output lines are always prepended with a '#[Out]# ' marker, so that the log remains valid Python code.

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 in their final form, converted into valid Python. For example, %Exit is logged as '_ip.magic("Exit")'. If the -r flag is given, all input is logged exactly as typed, with no transformations applied.

-t: put timestamps before each input line logged (these are put in comments).

magic_logstate (*parameter_s*='')

Print the status of the logging system.

magic_logstop (*parameter_s*='')

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.

magic_lsmagic (*parameter_s*='')

List currently available magic functions.

magic_macro (*parameter_s*='')

Define a set of input lines as a macro for future re-execution.

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 as the command line is used instead.

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 notation for indicating number ranges is: n1-n2 means 'use line numbers n1,...n2' (the endpoint is included). That is, '5-7' means using the lines numbered 5,6 and 7.

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 (%hist prints it):

```
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'.
```

For one-off cases which DON'T contain magic function calls in them you can obtain similar results by explicitly executing slices from your input history with:

```
In [60]: exec In[44:48]+In[49]
```

magic_magic (*parameter_s*='')

Print information about the magic function system.

Supported formats: -latex, -brief, -rest

magic_page (*parameter_s*='')

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.

magic_pdb (*parameter_s*='')

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 ipythonrc configuration file (the variable is called '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.

magic_pdef (*parameter_s*='', *namespaces*=None)

Print the definition header for any callable object.

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

magic_pdoc (*parameter_s*='', *namespaces*=None)

Print the docstring for an object.

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

magic_pfile (*parameter_s*='')

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.

magic_pinfo (*parameter_s*='', *namespaces*=None)

Provide detailed information about an object.

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

magic_pinfo2 (*parameter_s*='', *namespaces*=None)

Provide extra detailed information about an object.

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

magic_popd (*parameter_s*='')

Change to directory popped off the top of the stack.

magic_pprint (*parameter_s*='')

Toggle pretty printing on/off.

magic_profile (*parameter_s*='')

Print your currently active IPython profile.

magic_prun (*parameter_s=''*, *user_mode=1*, *opts=None*, *arg_lst=None*, *prog_ns=None*)

Run a statement through the python code profiler.

Usage: `%prun [options] statement`

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 namespaces which do not hold under IPython.

Options:

`-l <limit>`: you can place restrictions on what or how much of the profile gets printed. The limit value can be:

- 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, use a limit of 0.4 to see the topmost 40% only).

You can combine several limits with repeated use of the option. For example, `'-l __init__ -l 5'` will print only the topmost 5 lines of information about class constructors.

`-r`: return the `pstats.Stats` object generated by the profiling. This object has all the information about the profile in it, and you can later use it for further analysis or in other functions.

-s <key>: **sort profile by given key. You can provide more than one key** by using the option several times: `'-s key1 -s key2 -s key3...'`. The default sorting key is 'time'.

The following is copied verbatim from the profile documentation referenced below:

When more than one key is provided, additional keys are used as secondary criteria when there is equality in all keys selected before them.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

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

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between "nfl" and "stdname" is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order "20" "3" and "40". In contrast, "nfl" does a numeric compare of the line numbers. In fact, `sort_stats("nfl")` is the same as `sort_stats("name", "file", "line")`.

-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 understood by the `pstats` module, and is generated by a call to the `dump_stats()` method of profile objects. The profile is still shown on screen.

If you want to run complete programs under the profiler's control, use `'%run -p [prof_opts] filename.py [args to program]'` where `prof_opts` contains profiler specific options as described here.

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

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

magic_psearch (*parameter_s*='')

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:

-a: makes the pattern match even objects whose names start with a single underscore. These names are normally omitted from the search.

-i/-c: make the pattern case insensitive/sensitive. If neither of these options is given, the default is read from your `ipythonrc` file. The option name which sets this value is `'wildcards_case_sensitive'`. If this option is not specified in your `ipythonrc` 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

`magic_psource` (*parameter_s*='', *namespaces*=None)

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

`magic_pushd` (*parameter_s*='')

Place the current dir on stack and change directory.

Usage: `%pushd` [*'dirname'*]

`magic_pwd` (*parameter_s*='')

Return the current working directory path.

`magic_pycat` (*parameter_s*='')

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.

`magic_pylab` (*s*)

Load numpy and matplotlib to work interactively.

`%pylab` [*GUINAME*]

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

It will import at the top level numpy as np, pyplot as plt, matplotlib, pylab and mlab, as well as all names from numpy and pylab.

Parameters `guiname` : optional

One of the valid arguments to the `%gui` magic ('qt', 'wx', 'gtk' or 'tk'). If given, the corresponding Matplotlib backend is used, otherwise matplotlib's default (which you can override in your matplotlib config file) is used.

Examples

In this case, where the MPL default is TkAgg: In [2]: `%pylab`

Welcome to pylab, a matplotlib-based Python environment. Backend in use: TkAgg For more information, type `'help(pylab)'`.

But you can explicitly request a different backend: In [3]: `%pylab qt`

Welcome to pylab, a matplotlib-based Python environment. Backend in use: Qt4Agg For more information, type `'help(pylab)'`.

`magic_quickref` (*arg*)

Show a quick reference sheet

`magic_quit` (*parameter_s*='')

Exit IPython.

`magic_r` (*parameter_s*='')

Repeat previous input.

Note: Consider using the more powerfull `%rep` instead!

If given an argument, repeats the previous command which starts with the same string, otherwise it just repeats the previous input.

Shell escaped commands (with `!` as first character) are not recognized by this system, only pure python code and magic commands.

`magic_rehashx` (*parameter_s*='')

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

This version explicitly checks that every entry in `$PATH` is a file with execute access (`os.X_OK`), so it is much slower than `%rehash`.

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 `'exelcomlbat'`.

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

`magic_reload_ext` (*module_str*)

Reload an IPython extension by its module name.

`magic_reset` (*parameter_s*='')

Resets the namespace by removing all names defined by the user.

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

Parameters `-y` : force reset without asking for confirmation.

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 [10]: 'a' in _ip.user_ns Out[10]: False

magic_reset_selective (*parameter_s=''*)

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.

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']

magic_run (*parameter_s='', runner=None, file_finder=<function get_py_filename at 0x104ab4c08>*)

Run the named file inside IPython as a program.

Usage: %run [-n -i -t [-N<N>] -d [-b<N>] -p [profile options]] 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 use (unless `-p` is used, see below).

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.

Options:

`-n`: `__name__` is NOT set to `'__main__'`, but to the running file's name without extension (as python does under `import`). This allows running scripts and reloading the definitions in them without calling code protected by an `'if __name__ == "__main__":'` clause.

`-i`: run the file in IPython's namespace instead of an empty one. This is useful if you are experimenting with code written in a text editor which depends on variables defined interactively.

`-e`: ignore `sys.exit()` calls or `SystemExit` exceptions in the script being run. This is particularly useful if IPython is being used to run unittests, which always exit with a `sys.exit()` call. In such cases you are interested in the output of the test results, not in seeing a traceback of the unittest module.

`-t`: print timing information at the end of the run. IPython will give you an estimated CPU time consumption for your script, which under Unix uses the `resource` module to avoid the wraparound problems of `time.clock()`. Under Unix, an estimate of time spent on system tasks is also given (for Windows platforms this is reported as 0.0).

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 runUser : 0.910862 s, 0.1821724 s.System: 0.0 s, 0.0 s.
```

`-d`: run your program under the control of `pdb`, the Python debugger. This allows you to execute your program step by step, watch variables, etc. Internally, what IPython does is similar to calling:

```
pdb.run('execfile("YOURFILENAME")')
```

with a breakpoint set on line 1 of your file. You can change the line number for this automatic breakpoint to be `<N>` by using the `-bN` option (where `N` must be an integer). For example:

```
%run -d -b40 myscript
```

will set the first breakpoint at line 40 in `myscript.py`. Note that the first breakpoint must be set on a line which actually does something (not a comment or docstring) for it to stop execution.

When the `pdb` debugger starts, you will see a `(Pdb)` prompt. You must first enter `'c'` (without quotes) to start execution up to the first breakpoint.

Entering `'help'` gives information about the use of the debugger. You can easily see `pdb`'s full documentation with `"import pdb;pdb.help()"` at a prompt.

`-p`: run program under the control of the Python profiler module (which prints a detailed report of execution times, function calls, etc).

You can pass other options after `-p` which affect the behavior of the profiler itself. See the docs for `%prun` for details.

In this mode, the program's variables do NOT propagate back to the IPython interactive namespace (because they remain in the namespace where the profiler executes them).

Internally this triggers a call to `%prun`, see its documentation for details on the options available specifically for profiling.

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

`magic_save` (*parameter_s=''*)

Save a set of lines 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.

This function uses the same syntax as `%macro` for line extraction, but instead of creating a macro it saves the resulting string 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.

`magic_sc` (*parameter_s=''*)

Shell capture - execute a shell command and capture its output.

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 space-separated string. These are convenient, respectively, either for sequential processing or to be passed to a shell command.

For example:

all-random

```
# Capture into variable a In [1]: sc a=ls *py
# a is a string with embedded newlines In [2]: a Out[2]:
'setup.pynwin32_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.

magic_sx (*parameter_s=''*)

Shell execute - run a shell command and capture its output.

%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.

magic_tb (*s*)

Print the last traceback with the currently active exception mode.

See %xmode for changing exception reporting modes.

magic_time (*parameter_s=''*)

Time execution of a Python statement or expression.

The CPU and wall clock times are printed, and the value of the expression (if any) is returned. Note that under Win32, system time is always reported as 0, since it can not be measured.

This function provides very basic timing functionality. In Python 2.3, the `timeit` module offers more control and sophistication, so this could be rewritten to use it (patches welcome).

Some 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
```

magic_timeit (*parameter_s*='')

Time execution of a Python statement or expression

Usage: %timeit [-n<N> -r<R> [-tl-c]] statement

Time execution of a Python statement or expression using the timeit module.

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

Examples:

```
In [1]: %timeit pass 10000000 loops, best of 3: 53.3 ns per loop
```

```
In [2]: u = None
```

```
In [3]: %timeit u is None 10000000 loops, best of 3: 184 ns per loop
```

```
In [4]: %timeit -r 4 u == None 1000000 loops, best of 4: 242 ns per loop
```

```
In [5]: import time
```

```
In [6]: %timeit -n1 time.sleep(2) 1 loops, best of 3: 2 s per loop
```

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.

magic_unalias (*parameter_s*='')

Remove an alias

magic_unload_ext (*module_str*)

Unload an IPython extension by its module name.

magic_who (*parameter_s*='')

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.

magic_who_ls (*parameter_s*='')

Return a sorted list of all interactive variables.

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

magic_whos (*parameter_s*='')

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 and Numeric arrays, a summary with shape, number of elements, typecode and size in memory.
- Everything else: a string representation, snipping their middle if too long.

magic_xmode (*parameter_s*='')

Switch modes for the exception handlers.

Valid modes: Plain, Context and Verbose.

If called without arguments, acts as a toggle.

parse_options (*arg_str*, *opt_str*, **long_opts*, ***kw*)

Parse options passed to an argument string.

The interface is similar to that of `getopt()`, but it returns back a Struct with the options as keys and the stripped argument string still as a string.

`arg_str` is quoted as a true `sys.argv` vector by using `shlex.split`. This allows us to easily expand variables, glob files, quote arguments, etc.

Options: `-mode`: default 'string'. If given as 'list', the argument string is returned as a list (split on whitespace) instead of a string.

`-list_all`: put all option values in lists. Normally only options appearing more than once are put in a list.

`-posix` (True): whether to split the input line in POSIX mode or not, as per the conventions outlined in the `shlex` module from the standard library.

profile_missing_notice (*args, **kwargs)

8.26.3 Functions

`IPython.core.magic.compress_dhist` (dh)

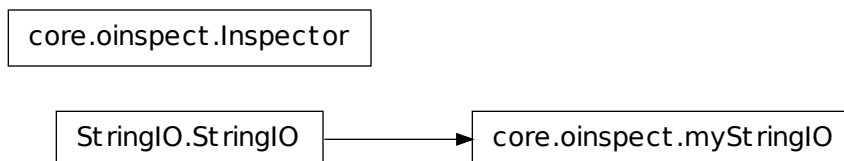
`IPython.core.magic.on_off` (tag)

Return an ON/OFF string for a 1/0 input. Simple utility function.

8.27 core.oinspect

8.27.1 Module: core.oinspect

Inheritance diagram for `IPython.core.oinspect`:



Tools for inspecting Python objects.

Uses syntax highlighting for presenting the various information elements.

Similar in spirit to the `inspect` module, but all calls take a name argument to reference the name under which an object is being read.

8.27.2 Classes

Inspector

```
class IPython.core.oinspect.Inspector (color_table={'':
    <IPython.utils.coloransi.ColorScheme instance at 0x103e43680>,
    'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x103e43170>,
    'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x103e433f8>,
    'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x103e43680>},
    code_color_table={'':
    <IPython.utils.coloransi.ColorScheme instance at 0x103ede5f0>,
    'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x103ede638>,
    'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x103ede128>,
    'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x103ede5f0>},
    scheme='NoColor',
    str_detail_level=0)
```

```
__init__(color_table={'': <IPython.utils.coloransi.ColorScheme instance at 0x103e43680>,
    'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x103e43170>,
    'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x103e433f8>,
    'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x103e43680>},
    code_color_table={'': <IPython.utils.coloransi.ColorScheme instance at 0x103ede5f0>,
    'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x103ede638>,
    'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x103ede128>,
    'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x103ede5f0>},
    scheme='NoColor', str_detail_level=0)
```

info (*obj*, *oname*='', *formatter*=None, *info*=None, *detail_level*=0)

Compute a dict with detailed information about an object.

Optional arguments:

- *oname*: name of the variable pointing to the object.
- *formatter*: special formatter for docstrings (see `pdoc`)
- *info*: a structure with some information fields which may have been

precomputed already.

- *detail_level*: if set to 1, more information is given.

noinfo (*msg*, *oname*)

Generic message when no information is found.

pdef (*obj*, *oname*='')

Print the definition header for any callable object.

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

pdoc (*obj*, *oname*='', *formatter*=None)

Print the docstring for any object.

Optional: -formatter: a function to run the docstring through for specially formatted docstrings.

pfile (*obj*, *oname*='')

Show the whole file where an object was defined.

pinfo (*obj*, *oname*='', *formatter*=None, *info*=None, *detail_level*=0)

Show detailed information about an object.

Optional arguments:

- oname: name of the variable pointing to the object.
- formatter: special formatter for docstrings (see pdoc)
- info: a structure with some information fields which may have been precomputed already.
- detail_level: if set to 1, more information is given.

psearch (*pattern*, *ns_table*, *ns_search*=[], *ignore_case*=False, *show_all*=False)

Search namespaces with wildcards for objects.

Arguments:

- pattern: string containing shell-like wildcards to use in namespace searches and optionally a type specification to narrow the search to objects of that type.
- ns_table: dict of name->namespaces for search.

Optional arguments:

- ns_search: list of namespace names to include in search.
- ignore_case(False): make the search case-insensitive.
- show_all(False): show all names, including those starting with underscores.

psource (*obj*, *oname*='')

Print the source code for an object.

set_active_scheme (*scheme*)

myStringIO

```
class IPython.core.oinspect.myStringIO (buf='')
```

```
    Bases: StringIO.StringIO
```

Adds a `writeln` method to normal `StringIO`.

`__init__` (*buf*='')

`close` ()

Free the memory buffer.

`flush` ()

Flush the internal buffer

`getvalue` ()

Retrieve the entire contents of the “file” at any time before the `StringIO` object’s `close()` method is called.

The `StringIO` object can accept either Unicode or 8-bit strings, but mixing the two may take some care. If both are used, 8-bit strings that cannot be interpreted as 7-bit ASCII (that use the 8th bit) will cause a `UnicodeError` to be raised when `getvalue()` is called.

`isatty` ()

Returns `False` because `StringIO` objects are not connected to a tty-like device.

`next` ()

A file object is its own iterator, for example `iter(f)` returns `f` (unless `f` is closed). When a file is used as an iterator, typically in a for loop (for example, for `line` in `f`: `print line`), the `next()` method is called repeatedly. This method returns the next input line, or raises `StopIteration` when EOF is hit.

`read` (*n=-1*)

Read at most `size` bytes from the file (less if the read hits EOF before obtaining `size` bytes).

If the `size` argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

`readline` (*length=None*)

Read one entire line from the file.

A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). If the `size` argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned.

An empty string is returned only when EOF is encountered immediately.

Note: Unlike `stdio`’s `fgets()`, the returned string contains null characters (`'0'`) if they occurred in the input.

`readlines` (*sizehint=0*)

Read until EOF using `readline()` and return a list containing the lines thus read.

If the optional `sizehint` argument is present, instead of reading up to EOF, whole lines totalling approximately `sizehint` bytes (or more to accommodate a final whole line).

`seek` (*pos, mode=0*)

Set the file’s current position.

The `mode` argument is optional and defaults to 0 (absolute file positioning); other values are 1 (seek relative to the current position) and 2 (seek relative to the file’s end).

There is no return value.

tell()

Return the file's current position.

truncate (*size=None*)

Truncate the file's size.

If the optional size argument is present, the file is truncated to (at most) that size. The size defaults to the current position. The current file position is not changed unless the position is beyond the new file size.

If the specified size exceeds the file's current size, the file remains unchanged.

write (*s*)

Write a string to the file.

There is no return value.

writelines (*iterable*)

Write a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings. There is no return value.

(The name is intended to match `readlines()`; `writelines()` does not add line separators.)

writeln (**arg, **kw*)

Does a `write()` and then a `write(' ')`

8.27.3 Functions

`IPython.core.oinspect.call_tip(oinfo, format_call=True)`

Extract call tip data from an oinfo dict.

Parameters **oinfo** : dict

format_call : bool, optional

If True, the call line is formatted and returned as a string. If not, a tuple of (name, argspec) is returned.

Returns **call_info** : None, str or (str, dict) tuple.

When `format_call` is True, the whole call information is formatted as a single string. Otherwise, the object's name and its argspec dict are returned. If no call information is available, None is returned.

docstring : str or None

The most relevant docstring for calling purposes is returned, if available. The priority is: call docstring for callable instances, then constructor docstring for classes, then main object's docstring otherwise (regular functions).

`IPython.core.oinspect.format_argspec(argspec)`

Format argspec, convenience wrapper around `inspect`'s.

This takes a dict instead of ordered arguments and calls `inspect.format_argspec` with the arguments in the necessary order.

`IPython.core.oinspect.getargspec(obj)`

Get the names and default values of a function's arguments.

A tuple of four things is returned: (args, varargs, varkw, defaults). 'args' is a list of the argument names (it may contain nested lists). 'varargs' and 'varkw' are the names of the * and ** arguments or None. 'defaults' is an n-tuple of the default values of the last n arguments.

Modified version of `inspect.getargspec` from the Python Standard Library.

`IPython.core.oinspect.getdoc(obj)`

Stable wrapper around `inspect.getdoc`.

This can't crash because of attribute problems.

It also attempts to call a `getdoc()` method on the given object. This allows objects which provide their docstrings via non-standard mechanisms (like Pyro proxies) to still be inspected by ipython's ? system.

`IPython.core.oinspect.getsource(obj, is_binary=False)`

Wrapper around `inspect.getsource`.

This can be modified by other projects to provide customized source extraction.

Inputs:

- obj: an object whose source code we will attempt to extract.

Optional inputs:

- is_binary: whether the object is known to come from a binary source.

This implementation will skip returning any output for binary objects, but custom extractors may know how to meaningfully process them.

`IPython.core.oinspect.object_info(**kw)`

Make an object info dict with all fields present.

8.28 core.page

8.28.1 Module: core.page

Paging capabilities for IPython.core

Authors:

- Brian Granger
- Fernando Perez

Notes

For now this uses `ipapi`, so it can't be in `IPython.utils`. If we can get rid of that dependency, we could move it there. —

8.28.2 Functions

`IPython.core.page.get_pager_cmd(pager_cmd=None)`

Return a pager command.

Makes some attempts at finding an OS-correct one.

`IPython.core.page.get_pager_start(pager, start)`

Return the string for paging files with an offset.

This is the '+N' argument which less and more (under Unix) accept.

`IPython.core.page.page(strng, start=0, screen_lines=0, pager_cmd=None)`

Print a string, piping through a pager after a certain length.

The `screen_lines` parameter specifies the number of *usable* lines of your terminal screen (total lines minus lines you need to reserve to show other information).

If you set `screen_lines` to a number ≤ 0 , `page()` will try to auto-determine your screen size and will only use up to $(\text{screen_size} + \text{screen_lines})$ for printing, paging after that. That is, if you want auto-detection but need to reserve the bottom 3 lines of the screen, use `screen_lines = -3`, and for auto-detection without any lines reserved simply use `screen_lines = 0`.

If a string won't fit in the allowed lines, it is sent through the specified pager command. If none given, look for `PAGER` in the environment, and ultimately default to `less`.

If no system pager works, the string is sent through a 'dumb pager' written in python, very simplistic.

`IPython.core.page.page_dumb(strng, start=0, screen_lines=25)`

Very dumb 'pager' in Python, for when nothing else works.

Only moves forward, same interface as `page()`, except for `pager_cmd` and `mode`.

`IPython.core.page.page_file(fname, start=0, pager_cmd=None)`

Page a file, using an optional pager command and starting line.

`IPython.core.page.snip_print(str, width=75, print_full=0, header='')`

Print a string snipping the midsection to fit in width.

print_full: mode control:

- 0: only snip long strings
- 1: send to `page()` directly.
- 2: snip long strings and ask for full length viewing with `page()`

Return 1 if snipping was necessary, 0 otherwise.

8.29 core.payload

8.29.1 Module: `core.payload`

Inheritance diagram for `IPython.core.payload`:



Payload system for IPython.

Authors:

- Fernando Perez
- Brian Granger

8.29.2 PayloadManager

```
class IPython.core.payload.PayloadManager(**kwargs)
    Bases: IPython.config.configurable.Configurable
```

```
__init__(**kwargs)
    Create a configurable given a config config.
```

Parameters `config`: Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

```
clear_payload()
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

read_payload ()

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

write_payload (*data*)

8.30 core.payloadpage

8.30.1 Module: core.payloadpage

A payload based version of page.

Authors:

- Brian Granger
- Fernando Perez

8.30.2 Functions

`IPython.core.payloadpage.install_payload_page()`

Install this version of page as `IPython.core.page.page`.

`IPython.core.payloadpage.page(strng, start=0, screen_lines=0, pager_cmd=None, html=None, auto_html=False)`

Print a string, piping through a pager.

This version ignores the `screen_lines` and `pager_cmd` arguments and uses IPython's payload system instead.

Parameters `strng` : str

Text to page.

start : int

Starting line at which to place the display.

html : str, optional

If given, an html string to send as well.

auto_html : bool, optional

If true, the input string is assumed to be valid reStructuredText and is converted to HTML with docutils. Note that if docutils is not found, this option is silently ignored.

8.31 core.plugin

8.31.1 Module: `core.plugin`

Inheritance diagram for `IPython.core.plugin`:



IPython plugins.

Authors:

- Brian Granger

8.31.2 Classes

Plugin

class IPython.core.plugin.**Plugin** (**kwargs)
Bases: IPython.config.configurable.Configurable

Base class for IPython plugins.

__init__ (**kwargs)
Create a configurable given a config config.

Parameters **config** : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the **__init__()** method of Configurable *before* doing anything else and using **super()**:

```
class MyConfigurable(Configurable):  
    def __init__(self, config=None):  
        super(MyConfigurable, self).__init__(config)  
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method **_a_changed(self, name, old, new)** (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be **handler()**, **handler(name)**, **handler(name, new)** or **handler(name, old, new)**.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

PluginManager

class IPython.core.plugin.**PluginManager** (*config=None*)

Bases: IPython.config.configurable.Configurable

A manager for IPython plugins.

__init__ (*config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

get_plugin (*name, default=None*)

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

plugins

An instance of a Python dict.

register_plugin (*name, plugin*)

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

unregister_plugin (*name*)

8.32 core.prefilter

8.32.1 Module: core.prefilter

Inheritance diagram for IPython.core.prefilter:



Prefiltering components.

Prefilters transform user input before it is exec'd by Python. These transforms are used to implement additional syntax such as `!ls` and `%magic`.

Authors:

- Brian Granger
- Fernando Perez
- Dan Milstein
- Ville Vainio

8.32.2 Classes

AliasChecker

```
class IPython.core.prefilter.AliasChecker (shell=None, prefilter_manager=None,
                                           config=None)
    Bases: IPython.core.prefilter.PrefilterChecker
    __init__ (shell=None, prefilter_manager=None, config=None)
```

check (line_info)

Check if the initial identifier on the line is an alias.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

AliasHandler

class IPython.core.prefilter.**AliasHandler** (*shell=None, prefilter_manager=None, config=None*)

Bases: IPython.core.prefilter.PrefilterHandler

__init__ (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Handle alias input lines.

handler_name

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

AssignMagicTransformer

```
class IPython.core.prefilter.AssignMagicTransformer (shell=None,          pre-
                                                    filter_manager=None,
                                                    config=None)
```

Bases: IPython.core.prefilter.PrefilterTransformer

Handle the *a = %who* syntax.

__init__ (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

transform (*line, continue_prompt*)

AssignSystemTransformer

```
class IPython.core.prefilter.AssignSystemTransformer (shell=None,      pre-
                                                    filter_manager=None,
                                                    config=None)
    Bases: IPython.core.prefilter.PrefilterTransformer
```

Handle the `files = !ls` syntax.

`__init__` (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

`on_trait_change` (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

`trait_metadata` (*traitname, key*)

Get metadata values for trait by key.

`trait_names` (***metadata*)

Get a list of all the names of this classes traits.

`traits` (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

transform (*line, continue_prompt*)

AssignmentChecker

```
class IPython.core.prefilter.AssignmentChecker (shell=None,           pre-
                                              filter_manager=None,      con-
                                              fig=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

Check to see if user is assigning to a var for the first time, in which case we want to avoid any sort of automagic / autocall games.

This allows users to assign to either alias or magic names true python variables (the magic/alias systems always take second seat to true python code). E.g. `ls='hi'`, or `ls,that=1,2`

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait `'a'`, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If `False` (the default), then install the handler. If `True` then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

AutoHandler

```
class IPython.core.prefilter.AutoHandler (shell=None,      prefilter_manager=None,
                                          config=None)
```

Bases: IPython.core.prefilter.PrefilterHandler

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Handle lines which can be auto-executed, quoting if requested.

handler_name

A trait for strings.

```
on_trait_change (handler, name=None, remove=False)
```

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstalls it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

AutoMagicChecker

```
class IPython.core.prefilter.AutoMagicChecker (shell=None,           pre-
                                              filter_manager=None,      con-
                                              fig=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

check (*line_info*)

If the ifun is magic, and automagic is on, run it. Note: normal, non-auto magic would already have been triggered via '%' in `check_esc_chars`. This just checks for automagic. Also, before triggering the magic handler, make sure that there is nothing in the user namespace which could shadow it.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

AutocallChecker

```
class IPython.core.prefilter.AutocallChecker(shell=None, filter_manager=None, fig=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

```
__init__(shell=None, prefilter_manager=None, config=None)
```

check (*line_info*)

Check if the initial word/function is callable and autocall is on.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

EmacsChecker

```
class IPython.core.prefilter.EmacsChecker (shell=None,    prefilter_manager=None,
                                           config=None)
```

Bases: IPython.core.prefilter.PrefilterChecker

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

check (*line_info*)

Emacs ipython-mode tags certain input lines.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unntall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

EmacsHandler

```
class IPython.core.prefilter.EmacsHandler (shell=None, prefilter_manager=None,
                                           config=None)
```

Bases: `IPython.core.prefilter.PrefilterHandler`

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Handle input lines marked by python-mode.

handler_name

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

EscCharsChecker

```
class IPython.core.prefilter.EscCharsChecker (shell=None, filter_manager=None, fig=None) pre-con-
```

Bases: `IPython.core.prefilter.PrefilterChecker`

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

Check for escape character and return either a handler to handle it, or None if there is no escape char.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

HelpHandler

class IPython.core.prefilter.**HelpHandler** (*shell=None, prefilter_manager=None, config=None*)

Bases: IPython.core.prefilter.PrefilterHandler

__init__ (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Try to get some help for the object.

obj? or ?obj -> basic information. obj?? or ??obj -> more details.

handler_name

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

IPyAutocallChecker

```
class IPython.core.prefilter.IPyAutocallChecker (shell=None,           pre-
                                                filter_manager=None,      con-
                                                fig=None)
```

Bases: IPython.core.prefilter.PrefilterChecker

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

check (*line_info*)

Instances of IPyAutocall in user_ns get autocalled immediately

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

IPythonPromptTransformer

```
class IPython.core.prefilter.IPythonPromptTransformer (shell=None,           pre-
                                                         filter_manager=None,
                                                         config=None)
Bases: IPython.core.prefilter.PrefilterTransformer
```

Handle inputs that start classic IPython prompt syntax.

__init__ (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

transform (*line*, *continue_prompt*)

LineInfo

class IPython.core.prefilter.**LineInfo** (*line*, *continue_prompt*)

Bases: object

A single line of input and associated info.

Includes the following as properties:

line The original, raw line

continue_prompt Is this line a continuation in a sequence of multiline input?

pre The initial esc character or whitespace.

pre_char The escape character(s) in pre or the empty string if there isn't one. Note that '!!' is a possible value for pre_char. Otherwise it will always be a single character.

pre_whitespace The leading whitespace from pre if it exists. If there is a pre_char, this is just ''.

ifun The 'function part', which is basically the maximal initial sequence of valid python identifiers and the '.' character. This is what is checked for alias and magic transformations, used for auto-calling, etc.

the_rest Everything else on the line.

__init__ (*line*, *continue_prompt*)

ofind (*ip*)

Do a full, attribute-walking lookup of the ifun in the various namespaces for the given IPython InteractiveShell instance.

Return a dict with keys: found,obj,ospace,ismagic

Note: can cause state changes because of calling getattr, but should only be run if autocall is on and if the line hasn't matched any other, less dangerous handlers.

Does cache the results of the call, so can be called multiple times without worrying about *further* damaging state.

MagicHandler

class IPython.core.prefilter.**MagicHandler** (*shell=None*, *prefilter_manager=None*,
config=None)

Bases: IPython.core.prefilter.PrefilterHandler

__init__ (*shell=None*, *prefilter_manager=None*, *config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Execute magic functions.

handler_name

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

MultiLineMagicChecker

```
class IPython.core.prefilter.MultiLineMagicChecker (shell=None,          pre-
                                                    filter_manager=None,
                                                    config=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

Allow `!` and `!!` in multi-line statements if `multi_line_specials` is on

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait `'a'`, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If `False` (the default), then install the handler. If `True` then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

PrefilterChecker

```
class IPython.core.prefilter.PrefilterChecker (shell=None,           pre-
                                              filter_manager=None,    con-
                                              fig=None)
```

Bases: IPython.config.configurable.Configurable

Inspect an input line and return a handler for that line.

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

check (*line_info*)

Inspect line_info and return a handler instance or None.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn't exist.

PrefilterError

```
class IPython.core.prefilter.PrefilterError
```

```
    Bases: exceptions.Exception
```

```
    __init__()
```

```
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

```
    args
```

```
    message
```


PrefilterHandler

```
class IPython.core.prefilter.PrefilterHandler (shell=None,           pre-
                                              filter_manager=None,    con-
                                              fig=None)
```

Bases: IPython.config.configurable.Configurable

__init__ (shell=None, prefilter_manager=None, config=None)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (line_info)

Handle normal input lines. Use as a template for handlers.

handler_name

A trait for strings.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

PrefilterManager

class IPython.core.prefilter.**PrefilterManager** (*shell=None, config=None*)

Bases: IPython.config.configurable.Configurable

Main prefilter component.

The IPython prefilter is run on all user input before it is run. The prefilter consumes lines of input and produces transformed lines of input.

The implementation consists of two phases:

- 1.Transformers
- 2.Checkers and handlers

Over time, we plan on deprecating the checkers and handlers and doing everything in the transformers.

The transformers are instances of `PrefilterTransformer` and have a single method `transform()` that takes a line and returns a transformed line. The transformation can be accomplished using any tool, but our current ones use regular expressions for speed. We also ship `pyarsing` in `IPython.external` for use in transformers.

After all the transformers have been run, the line is fed to the checkers, which are instances of `PrefilterChecker`. The line is passed to the `check()` method, which either returns `None` or a `PrefilterHandler` instance. If `None` is returned, the other checkers are tried. If an `PrefilterHandler` instance is returned, the line is passed to the `handle()` method of the returned handler and no further checkers are tried.

Both transformers and checkers have a *priority* attribute, that determines the order in which they are called. Smaller priorities are tried first.

Both transformers and checkers also have *enabled* attribute, which is a boolean that determines if the instance is used.

Users or developers can change the priority or enabled attribute of transformers or checkers, but they must call the `sort_checkers()` or `sort_transformers()` method after changing the priority.

__init__ (*shell=None, config=None*)

checkers

Return a list of checkers, sorted by priority.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

find_handler (*line_info*)

Find a handler for the *line_info* by trying checkers.

get_handler_by_esc (*esc_str*)

Get a handler by its escape string.

get_handler_by_name (*name*)

Get a handler by its name.

handlers

Return a dict of all the handlers.

init_checkers ()

Create the default checkers.

init_handlers ()

Create the default handlers.

init_transformers ()

Create the default transformers.

multi_line_specials

A casting version of the boolean trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘*_[traitname]_changed*’. Thus, to create static handler for the trait ‘a’, create the method *_a_changed*(self, name, old, new) (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be *handler()*, *handler(name)*, *handler(name, new)* or *handler(name, old, new)*.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_line (*line, continue_prompt=False*)

Prefilter a single input line as text.

This method prefilters a single line of text by calling the transformers and then the checkers/handlers.

prefilter_line_info (*line_info*)

Prefilter a line that has been converted to a LineInfo object.

This implements the checker/handler part of the prefilter pipe.

prefilter_lines (*lines*, *continue_prompt=False*)

Prefilter multiple input lines of text.

This is the main entry point for prefiltering multiple lines of input. This simply calls `prefilter_line()` for each line of input.

This covers cases where there are multiple lines in the user entry, which is the case when the user goes back to a multiline history entry and presses enter.

register_checker (*checker*)

Register a checker instance.

register_handler (*name*, *handler*, *esc_strings*)

Register a handler instance by name with `esc_strings`.

register_transformer (*transformer*)

Register a transformer instance.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

sort_checkers ()

Sort the checkers by priority.

This must be called after the priority of a checker is changed. The `register_checker()` method calls this automatically.

sort_transformers ()

Sort the transformers by priority.

This must be called after the priority of a transformer is changed. The `register_transformer()` method calls this automatically.

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns `None` if a metadata key doesn't exist.

transform_line (*line, continue_prompt*)

Calls the enabled transformers in order of increasing priority.

transformers

Return a list of checkers, sorted by priority.

unregister_checker (*checker*)

Unregister a checker instance.

unregister_handler (*name, handler, esc_strings*)

Unregister a handler instance by name with `esc_strings`.

unregister_transformer (*transformer*)

Unregister a transformer instance.

PrefilterTransformer

```
class IPython.core.prefilter.PrefilterTransformer (shell=None,           pre-
                                                filter_manager=None,
                                                config=None)
```

Bases: `IPython.config.configurable.Configurable`

Transform a line of user input.

__init__ (*shell=None, prefilter_manager=None, config=None*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait `'a'`, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

transform (*line, continue_prompt*)

Transform a line, returning the new one.

PyPromptTransformer

```
class IPython.core.prefilter.PyPromptTransformer (shell=None,           pre-
                                                filter_manager=None,    con-
                                                fig=None)
```

Bases: IPython.core.prefilter.PrefilterTransformer

Handle inputs that start with '>>>' syntax.

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

transform (*line, continue_prompt*)

PythonOpsChecker

```
class IPython.core.prefilter.PythonOpsChecker (shell=None,          pre-
                                              filter_manager=None,    con-
                                              fig=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

If the ‘rest’ of the line begins with a function call or pretty much any python operator, we should simply execute the line (regardless of whether or not there’s a possible autocall expansion). This avoids spurious (and very confusing) `getattr()` accesses.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a `HasTraits` subclass with the naming convention ‘`_[traitname]_changed`’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters handler : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

ShellEscapeChecker

```
class IPython.core.prefilter.ShellEscapeChecker (shell=None,           pre-
                                              filter_manager=None,      con-
                                              fig=None)
```

Bases: `IPython.core.prefilter.PrefilterChecker`

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

enabled

A boolean (True, False) trait.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention `'_[traitname]_changed'`. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

ShellEscapeHandler

```
class IPython.core.prefilter.ShellEscapeHandler (shell=None, filter_manager=None, fig=None)
```

Bases: `IPython.core.prefilter.PrefilterHandler`

```
__init__ (shell=None, prefilter_manager=None, config=None)
```

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

esc_strings

An instance of a Python list.

handle (*line_info*)

Execute the line in a shell, empty return value

handler_name

A trait for strings.

on_trait_change (*handler, name=None, remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then uninstall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because `get_metadata` returns None if a metadata key doesn’t exist.

8.32.3 Function

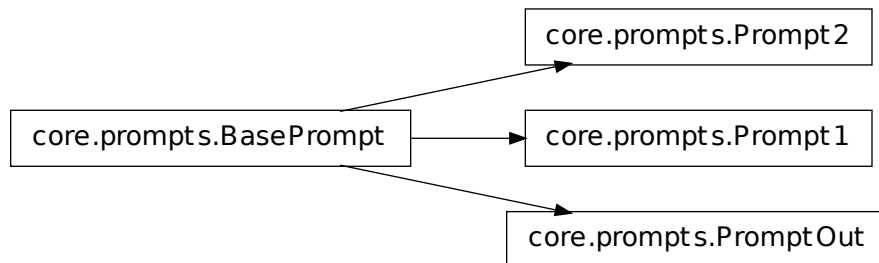
`IPython.core.prefilter.is_shadowed` (*identifier, ip*)

Is the given identifier defined in one of the namespaces which shadow the alias and magic namespaces? Note that an identifier is different than ifun, because it can not contain a ‘.’ character.

8.33 core.prompts

8.33.1 Module: `core.prompts`

Inheritance diagram for `IPython.core.prompts`:



Classes for handling input/output prompts.

Authors:

- Fernando Perez
- Brian Granger

8.33.2 Classes

BasePrompt

class `IPython.core.prompts.BasePrompt` (*cache, sep, prompt, pad_left=False*)

Bases: `object`

Interactive prompt similar to Mathematica's.

__init__ (*cache, sep, prompt, pad_left=False*)

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

```
set_p_str()
```

Set the interpolating prompt strings.

This must be called every time the color settings change, because the `prompt_specials` global may have changed.

```
write(msg)
```

Prompt1

```
class IPython.core.prompts.Prompt1(cache, sep='n', prompt='In [#]: ', pad_left=True)
```

Bases: `IPython.core.prompts.BasePrompt`

Input interactive prompt similar to Mathematica's.

```
__init__(cache, sep='n', prompt='In [#]: ', pad_left=True)
```

```
auto_rewrite()
```

Return a string of the form `'—>'` which lines up with the previous input string. Useful for systems which re-write the user input when handling automatically special syntaxes.

```
cwd_filt(depth)
```

Return the last depth elements of the current working directory.

\$HOME is always replaced with `'~'`. If `depth==0`, the full path is returned.

```
cwd_filt2(depth)
```

Return the last depth elements of the current working directory.

\$HOME is always replaced with `'~'`. If `depth==0`, the full path is returned.

```
p_template
```

Template for prompt string creation

```
set_colors()
```

```
set_p_str()
```

Set the interpolating prompt strings.

This must be called every time the color settings change, because the `prompt_specials` global may have changed.

```
write(msg)
```

Prompt2

```
class IPython.core.prompts.Prompt2(cache, prompt='>.\D.: ', pad_left=True)
```

Bases: `IPython.core.prompts.BasePrompt`

Interactive continuation prompt.

```
__init__(cache, prompt='>.\D.: ', pad_left=True)
```

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors ()

set_p_str ()

write (*msg*)

PromptOut

```
class IPython.core.prompts.PromptOut (cache, sep=',', prompt='Out[#]: ',  
                                       pad_left=True)
```

Bases: `IPython.core.prompts.BasePrompt`

Output interactive prompt similar to Mathematica's.

```
__init__ (cache, sep=',', prompt='Out[#]: ', pad_left=True)
```

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors ()

set_p_str ()

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write (*msg*)

8.33.3 Functions

`IPython.core.prompts.multiple_replace (dict, text)`

Replace in ‘text’ all occurrences of any key in the given dictionary by its corresponding value. Returns the new string.

`IPython.core.prompts.str_safe (arg)`

Convert to a string, without ever raising an exception.

If `str(arg)` fails, `<ERROR: ... >` is returned, where ... is the exception error message.

8.34 `core.splitinput`

8.34.1 Module: `core.splitinput`

Simple utility for splitting user input.

Authors:

- Brian Granger
- Fernando Perez

`IPython.core.splitinput.split_user_input (line, pattern=None)`

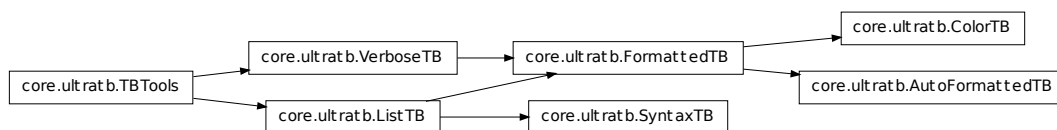
Split user input into pre-char/whitespace, function part and rest.

This is currently handles lines with ‘=’ in them in a very inconsistent manner.

8.35 `core.ultratb`

8.35.1 Module: `core.ultratb`

Inheritance diagram for `IPython.core.ultratb`:



`ultratb.py` – Spice up your tracebacks!

- ColorTB

I’ve always found it a bit hard to visually parse tracebacks in Python. The `ColorTB` class is a solution to that problem. It colors the different parts of a traceback in a manner similar to what you would expect from a syntax-highlighting text editor.

Installation instructions for ColorTB: `import sys,ultratb sys.excepthook = ultratb.ColorTB()`

- VerboseTB

I've also included a port of Ka-Ping Yee's "cgitb.py" that produces all kinds of useful info when a traceback occurs. Ping originally had it spit out HTML and intended it for CGI programmers, but why should they have all the fun? I altered it to spit out colored text to the terminal. It's a bit overwhelming, but kind of neat, and maybe useful for long-running programs that you believe are bug-free. If a crash *does* occur in that type of program you want details. Give it a shot—you'll love it or you'll hate it.

Note:

The Verbose mode prints the variables currently visible where the exception happened (shortening their strings if too long). This can potentially be very slow, if you happen to have a huge data structure whose string representation is complex to compute. Your computer may appear to freeze for a while with cpu usage at 100%. If this occurs, you can cancel the traceback with Ctrl-C (maybe hitting it more than once).

If you encounter this kind of situation often, you may want to use the Verbose_novars mode instead of the regular Verbose, which avoids formatting variables (but otherwise includes the information and context given by Verbose).

Installation instructions for ColorTB: `import sys,ultratb sys.excepthook = ultratb.VerboseTB()`

Note: Much of the code in this module was lifted verbatim from the standard library module 'traceback.py' and Ka-Ping Yee's 'cgitb.py'.

- Color schemes

The colors are defined in the class TBTools through the use of the ColorSchemeTable class. Currently the following exist:

- NoColor: allows all of this module to be used in any terminal (the color escapes are just dummy blank strings).
- Linux: is meant to look good in a terminal like the Linux console (black or very dark background).
- LightBG: similar to Linux but swaps dark/light colors to be more readable in light background terminals.

You can implement other color schemes easily, the syntax is fairly self-explanatory. Please send back new schemes you develop to the author for possible inclusion in future releases.

8.35.2 Classes

AutoFormattedTB

```
class IPython.core.ultratb.AutoFormattedTB (mode='Plain', color_scheme='Linux',  
                                           call_pdb=False, ostream=None,  
                                           tb_offset=0, long_header=False,  
                                           include_vars=False,  
                                           check_cache=None)
```

Bases: `IPython.core.ultratb.FormattedTB`

A traceback printer which can be called on the fly.

It will find out about exceptions by itself.

A brief example:

AutoTB = AutoFormattedTB(mode = 'Verbose',color_scheme='Linux') try:

...

except: AutoTB() # or AutoTB(out=logfile) where logfile is an open file object

```
__init__ (mode='Plain', color_scheme='Linux', call_pdb=False, ostream=None,  
          tb_offset=0, long_header=False, include_vars=False, check_cache=None)
```

```
color_toggle ()
```

Toggle between the currently active color scheme and NoColor.

```
context ()
```

```
debugger (force=False)
```

Call up the pdb debugger if desired, always clean up the tb reference.

Keywords:

- force(False):** by default, this routine checks the instance `call_pdb`

flag and does not actually invoke the debugger if the flag is false. The 'force' option forces the debugger to activate even if the flag is false.

If the `call_pdb` flag is set, the pdb interactive debugger is invoked. In all cases, the `self.tb` reference to the current traceback is deleted to prevent lingering references which hamper memory management.

Note that each call to `pdb()` does an 'import readline', so if your app requires a special setup for the readline completers, you'll have to fix that by hand after invoking the exception handler.

```
get_exception_only (etype, value)
```

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

```
handler (info=None)
```

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve to `io.Term.cout`. This ensures compatibility with most tools, including Windows (where plain `stdout` doesn't recognize ANSI escapes).
- Any object with 'write' and 'flush' attributes.

plain()**set_colors(*args, **kw)**

Shorthand access to the color table scheme selector method.

set_mode(mode=None)

Switch to the desired mode.

If mode is not specified, cycles through the available modes.

show_exception_only(etype, evalue)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

stb2text(stb)

Convert a structured traceback (a list) to a string.

structured_traceback(etype=None, value=None, tb=None, tb_offset=None, context=5)**text(etype, value, tb, tb_offset=None, context=5)**

Return formatted traceback.

Subclasses may override this if they add extra arguments.

verbose()**ColorTB****class** IPython.core.ultratb.ColorTB(color_scheme='Linux', call_pdb=0)

Bases: IPython.core.ultratb.FormattedTB

Shorthand to initialize a FormattedTB in Linux colors mode.

__init__(color_scheme='Linux', call_pdb=0)**color_toggle()**

Toggle between the currently active color scheme and NoColor.

context()

debugger (*force=False*)

Call up the pdb debugger if desired, always clean up the tb reference.

Keywords:

- force(False)**: by default, this routine checks the instance `call_pdb`

flag and does not actually invoke the debugger if the flag is false. The ‘force’ option forces the debugger to activate even if the flag is false.

If the `call_pdb` flag is set, the pdb interactive debugger is invoked. In all cases, the `self.tb` reference to the current traceback is deleted to prevent lingering references which hamper memory management.

Note that each call to `pdb()` does an ‘import readline’, so if your app requires a special setup for the readline completers, you’ll have to fix that by hand after invoking the exception handler.

get_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

handler (*info=None*)

ostream

Output stream that exceptions are written to.

Valid values are:

- None**: the default, which means that IPython will dynamically resolve

to `io.Term.cout`. This ensures compatibility with most tools, including Windows (where plain stdout doesn’t recognize ANSI escapes).

- Any object with ‘write’ and ‘flush’ attributes.

plain ()

set_colors (**args, **kw*)

Shorthand access to the color table scheme selector method.

set_mode (*mode=None*)

Switch to the desired mode.

If mode is not specified, cycles through the available modes.

show_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype, value, tb, tb_offset=None, context=5*)

text (*etype, value, tb, tb_offset=None, context=5*)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

verbose ()

FormattedTB

```
class IPython.core.ultratb.FormattedTB (mode='Plain',          color_scheme='Linux',
                                       call_pdb=False, ostream=None, tb_offset=0,
                                       long_header=False,    include_vars=False,
                                       check_cache=None)
```

Bases: `IPython.core.ultratb.VerboseTB`, `IPython.core.ultratb.ListTB`

Subclass ListTB but allow calling with a traceback.

It can thus be used as a `sys.excepthook` for Python > 2.1.

Also adds ‘Context’ and ‘Verbose’ modes, not available in ListTB.

Allows a `tb_offset` to be specified. This is useful for situations where one needs to remove a number of topmost frames from the traceback (such as occurs with python programs that themselves execute other python code, like Python shells).

```
__init__ (mode='Plain',    color_scheme='Linux',    call_pdb=False,    ostream=None,
          tb_offset=0, long_header=False, include_vars=False, check_cache=None)
```

color_toggle ()

Toggle between the currently active color scheme and NoColor.

context ()

debugger (*force=False*)

Call up the pdb debugger if desired, always clean up the `tb` reference.

Keywords:

- force**(False): by default, this routine checks the instance `call_pdb`

flag and does not actually invoke the debugger if the flag is false. The ‘force’ option forces the debugger to activate even if the flag is false.

If the `call_pdb` flag is set, the pdb interactive debugger is invoked. In all cases, the `self.tb` reference to the current traceback is deleted to prevent lingering references which hamper memory management.

Note that each call to `pdb()` does an ‘import readline’, so if your app requires a special setup for the readline completers, you’ll have to fix that by hand after invoking the exception handler.

get_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

handler (*info=None*)

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve to `io.Term.cout`. This ensures compatibility with most tools, including Windows (where plain `stdout` doesn't recognize ANSI escapes).
- Any object with 'write' and 'flush' attributes.

plain ()

set_colors (**args, **kw*)

Shorthand access to the color table scheme selector method.

set_mode (*mode=None*)

Switch to the desired mode.

If mode is not specified, cycles through the available modes.

show_exception_only (*etype, evalue*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype, value, tb, tb_offset=None, context=5*)

text (*etype, value, tb, tb_offset=None, context=5*)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

verbose ()

ListTB

class IPython.core.ultratb.**ListTB** (*color_scheme='NoColor', call_pdb=False, ostream=None*)

Bases: IPython.core.ultratb.TBTools

Print traceback information from a traceback list, with optional color.

Calling: requires 3 arguments: (*etype, evalue, elist*)

as would be obtained by: *etype, evalue, tb* = `sys.exc_info()` if *tb*:

elist = `traceback.extract_tb(tb)`

else: *elist* = None

It can thus be used by programs which need to process the traceback before printing (such as console replacements based on the code module from the standard library).

Because they are meant to be called without a full traceback (only a list), instances of this class can't call the interactive pdb debugger.

__init__ (*color_scheme='NoColor', call_pdb=False, ostream=None*)

color_toggle ()

Toggle between the currently active color scheme and NoColor.

get_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve to io.Term.cout. This ensures compatibility with most tools, including Windows (where plain stdout doesn't recognize ANSI escapes).
- Any object with 'write' and 'flush' attributes.

set_colors (**args, **kw*)

Shorthand access to the color table scheme selector method.

show_exception_only (*etype, evalue*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype, value, elist, tb_offset=None, context=5*)

Return a color formatted string with the traceback info.

Parameters **etype** : exception type

Type of the exception raised.

value : object

Data stored in the exception

elist : list

List of frames, see class docstring for details.

tb_offset : int, optional

Number of frames in the traceback to skip. If not given, the instance value is used (set in constructor).

context : int, optional

Number of lines of context information to print.

Returns String with formatted exception. :

text (*etype, value, tb, tb_offset=None, context=5*)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

SyntaxTB

class IPython.core.ultratb.**SyntaxTB** (*color_scheme='NoColor'*)

Bases: IPython.core.ultratb.ListTB

Extension which holds some state: the last exception value

__init__ (*color_scheme='NoColor'*)

clear_err_state ()

Return the current error state and clear it

color_toggle ()

Toggle between the currently active color scheme and NoColor.

get_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve

to io.Term.cout. This ensures compatibility with most tools, including Windows (where plain stdout doesn't recognize ANSI escapes).

- Any object with 'write' and 'flush' attributes.

set_colors (**args, **kw*)

Shorthand access to the color table scheme selector method.

show_exception_only (*etype, value*)

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype, value, elist, tb_offset=None, context=5*)

Return a color formatted string with the traceback info.

Parameters **etype** : exception type

Type of the exception raised.

value : object

Data stored in the exception

elist : list

List of frames, see class docstring for details.

tb_offset : int, optional

Number of frames in the traceback to skip. If not given, the instance value is used (set in constructor).

context : int, optional

Number of lines of context information to print.

Returns **String with formatted exception.** :

text (*etype, value, tb, tb_offset=None, context=5*)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

TBTools

class IPython.core.ultratb.**TBTools** (*color_scheme='NoColor', call_pdb=False, ostream=None*)

Bases: object

Basic tools used by all traceback printer classes.

__init__ (*color_scheme='NoColor', call_pdb=False, ostream=None*)

color_toggle ()

Toggle between the currently active color scheme and NoColor.

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve to io.Term.cout. This ensures compatibility with most tools, including Windows (where plain stdout doesn't recognize ANSI escapes).
- Any object with 'write' and 'flush' attributes.

set_colors (*args, **kw)

Shorthand access to the color table scheme selector method.

stb2text (stb)

Convert a structured traceback (a list) to a string.

structured_traceback (etype, evalue, tb, tb_offset=None, context=5, mode=None)

Return a list of traceback frames.

Must be implemented by each class.

text (etype, value, tb, tb_offset=None, context=5)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

VerboseTB

class IPython.core.ultratb.VerboseTB (color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0, long_header=False, include_vars=True, check_cache=None)

Bases: IPython.core.ultratb.TBTools

A port of Ka-Ping Yee's cgib.py module that outputs color text instead of HTML. Requires inspect and pydoc. Crazy, man.

Modified version which optionally strips the topmost entries from the traceback, to be used with alternate interpreters (because their own code would appear in the traceback).

__init__ (color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0, long_header=False, include_vars=True, check_cache=None)

Specify traceback offset, headers and color scheme.

Define how many frames to drop from the tracebacks. Calling it with tb_offset=1 allows use of this handler in interpreters which will have their own code at the top of the traceback (VerboseTB will first remove that frame before printing the traceback info).

color_toggle ()

Toggle between the currently active color scheme and NoColor.

debugger (force=False)

Call up the pdb debugger if desired, always clean up the tb reference.

Keywords:

- force(False): by default, this routine checks the instance call_pdb

flag and does not actually invoke the debugger if the flag is false. The 'force' option forces the debugger to activate even if the flag is false.

If the call_pdb flag is set, the pdb interactive debugger is invoked. In all cases, the self.tb reference to the current traceback is deleted to prevent lingering references which hamper memory management.

Note that each call to `pdb()` does an ‘import readline’, so if your app requires a special setup for the readline completers, you’ll have to fix that by hand after invoking the exception handler.

handler (*info=None*)

ostream

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve to `io.Term.cout`. This ensures compatibility with most tools, including Windows (where plain `stdout` doesn’t recognize ANSI escapes).
- Any object with ‘write’ and ‘flush’ attributes.

set_colors (**args, **kw*)

Shorthand access to the color table scheme selector method.

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype, eval, etb, tb_offset=None, context=5*)

Return a nice text document describing the traceback.

text (*etype, value, tb, tb_offset=None, context=5*)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

8.35.3 Functions

`IPython.core.ultratb.findsource` (*object*)

Return the entire source file and starting line number for an object.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of all the lines in the file and the line number indexes a line in that list. An `IOError` is raised if the source code cannot be retrieved.

FIXED version with which we monkeypatch the `stdlib` to work around a bug.

`IPython.core.ultratb.fix_frame_records_filenames` (*records*)

Try to fix the filenames in each record from `inspect.getinnerframes()`.

Particularly, modules loaded from within zip files have useless filenames attached to their code object, and `inspect.getinnerframes()` just uses it.

`IPython.core.ultratb.inspect_error` ()

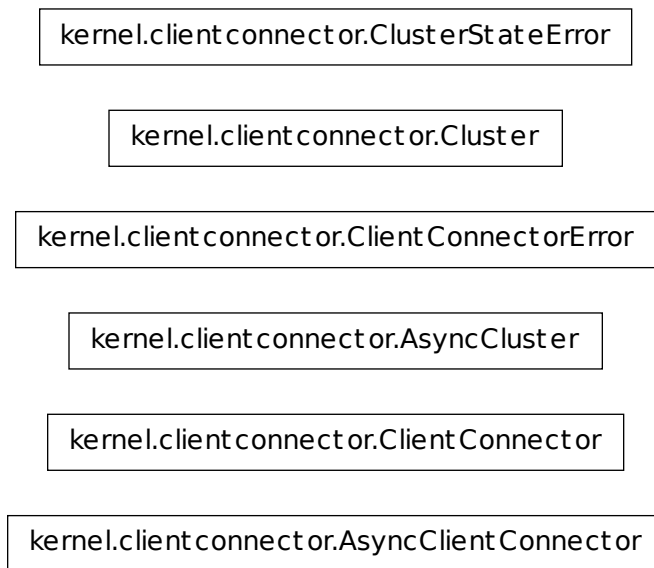
Print a message about internal inspect errors.

These are unfortunately quite common.

8.36 kernel.clientconnector

8.36.1 Module: `kernel.clientconnector`

Inheritance diagram for `IPython.kernel.clientconnector`:



Facilities for handling client connections to the controller.

8.36.2 Classes

`AsyncClientConnector`

class `IPython.kernel.clientconnector.AsyncClientConnector`

Bases: `object`

A class for getting remote references and clients from furls.

This start a single `Tub` for all remote reference and caches references.

`__init__()`

`get_client` (*profile='default', cluster_dir=None, furl_or_file=None, furl_file_name=None, ipython_dir=None, delay=0.20000000000000001, max_tries=9*)

Get a remote reference and wrap it in a client by furl.

This method is a simple wrapper around `get_client` that passes in the default name of the task client FURL file. Usually only the `profile` option will be needed. If a FURL file can't be

found by its profile, use `cluster_dir` or `furl_or_file`.

Parameters `profile` : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **:env:IPCLUSTER_DIR_PATH** environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

furl_or_file : str

A furl or a filename containing a FURL. This is useful if you simply know the location of the FURL file.

furl_file_name : str

The filename (not the full path) of the FURL. This must be provided if `furl_or_file` is not.

ipython_dir : str

The location of the `ipython_dir` if different from the default. This is used if the cluster directory is being found by profile.

delay : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

Returns A deferred to the actual client class. Or a failure to a :

:exc:'FURLError'. :

```
get_multiengine_client (profile='default', cluster_dir=None, furl_or_file=None,  
                        ipython_dir=None, delay=0.20000000000000001,  
                        max_tries=9)
```

Get the multiengine controller client.

This method is a simple wrapper around `get_client` that passes in the default name of the task client FURL file. Usually only the `profile` option will be needed. If a FURL file can't be found by its profile, use `cluster_dir` or `furl_or_file`.

Parameters `profile` : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **:env:IPCLUSTER_DIR_PATH** environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

furl_or_file : str

A furl or a filename containing a FURL. This is useful if you simply know the location of the FURL file.

ipython_dir : str

The location of the ipython_dir if different from the default. This is used if the cluster directory is being found by profile.

delay : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

Returns A deferred to the actual client class. :

get_reference (*furl_or_file*)

Get a remote reference using a furl or a file containing a furl.

Remote references are cached locally so once a remote reference has been retrieved for a given furl, the cached version is returned.

Parameters **furl_or_file** : str

A furl or a filename containing a furl. This should already be validated, but might not yet exist.

Returns A deferred to a remote reference :

get_task_client (*profile='default', cluster_dir=None, furl_or_file=None, ipython_dir=None, delay=0.20000000000000001, max_tries=9*)

Get the task controller client.

This method is a simple wrapper around `get_client` that passes in the default name of the task client FURL file. Usually only the `profile` option will be needed. If a FURL file can't be found by its profile, use `cluster_dir` or `furl_or_file`.

Parameters **profile** : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **env:** 'IPCLUSTER_DIR_PATH' environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

furl_or_file : str

A furl or a filename containing a FURL. This is useful if you simply know the location of the FURL file.

ipython_dir : str

The location of the `ipython_dir` if different from the default. This is used if the cluster directory is being found by profile.

delay : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

Returns A deferred to the actual client class. :

AsyncCluster

```
class IPython.kernel.clientconnector.AsyncCluster(profile='default',
                                                  cluster_dir=None,
                                                  ipython_dir=None,
                                                  auto_create=False,
                                                  auto_stop=True)
```

Bases: object

An class that wraps the **ipcluster** script.

```
__init__(profile='default', cluster_dir=None, ipython_dir=None, auto_create=False,
         auto_stop=True)
```

Create a class to manage an IPython cluster.

This class calls the **ipcluster** command with the right options to start an IPython cluster. Typically a cluster directory must be created (**ipcluster create**) and configured before using this class. Configuration is done by editing the configuration files in the top level of the cluster directory.

Parameters **profile** : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **env:** **IPCLUSTER_DIR_PATH** environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

ipython_dir : str

The location of the `ipython_dir` if different from the default. This is used if the cluster directory is being found by profile.

auto_create : bool

Automatically create the cluster directory if it doesn't exist. This will usually only make sense if using a local cluster (default=False).

auto_stop : bool

Automatically stop the cluster when this instance is garbage collected (default=True). This is useful if you want the cluster to live beyond your current process. There is also an instance attribute `auto_stop` to change this behavior.

get_ipcluster_logs ()

get_ipcontroller_logs ()

get_ipengine_logs ()

get_logs ()

get_logs_by_name (name='ipcluster')

get_multiengine_client (delay=0.20000000000000001, max_tries=9)

Get the multiengine client for the running cluster.

If this fails, it means that the cluster has not finished starting. Usually waiting a few seconds and re-trying will solve this.

get_task_client (delay=0.20000000000000001, max_tries=9)

Get the task client for the running cluster.

If this fails, it means that the cluster has not finished starting. Usually waiting a few seconds and re-trying will solve this.

location

running

start (*args, **kwargs)

stop (*args, **kwargs)

ClientConnector

class IPython.kernel.clientconnector.**ClientConnector**

Bases: object

A blocking version of a client connector.

This class creates a single `Tub` instance and allows remote references and client to be retrieved by their FURLs. Remote references are cached locally and FURL files can be found using profiles and cluster directories.

__init__ ()

```
get_client (profile='default', cluster_dir=None, furl_or_file=None, ipython_dir=None,  
            delay=0.20000000000000001, max_tries=9)
```

```
get_multiengine_client (profile='default', cluster_dir=None, furl_or_file=None,  
                          ipython_dir=None, delay=0.20000000000000001,  
                          max_tries=9)
```

Get the multiengine client.

Usually only the `profile` option will be needed. If a FURL file can't be found by its profile, use `cluster_dir` or `furl_or_file`.

Parameters `profile` : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **env:** `'IPCLUSTER_DIR_PATH'` environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

furl_or_file : str

A furl or a filename containing a FURL. This is useful if you simply know the location of the FURL file.

ipython_dir : str

The location of the `ipython_dir` if different from the default. This is used if the cluster directory is being found by profile.

delay : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

Returns The multiengine client instance. :

```
get_task_client (profile='default', cluster_dir=None, furl_or_file=None,  
                  ipython_dir=None, delay=0.20000000000000001, max_tries=9)
```

Get the task client.

Usually only the `profile` option will be needed. If a FURL file can't be found by its profile, use `cluster_dir` or `furl_or_file`.

Parameters `profile` : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **env:** `'IPCLUSTER_DIR_PATH'` environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

furl_or_file : str

A furl or a filename containing a FURL. This is useful if you simply know the location of the FURL file.

ipython_dir : str

The location of the ipython_dir if different from the default. This is used if the cluster directory is being found by profile.

delay : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

Returns The task client instance. :

ClientConnectorError

class IPython.kernel.clientconnector.**ClientConnectorError**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

Cluster

class IPython.kernel.clientconnector.**Cluster**(*profile='default', cluster_dir=None, ipython_dir=None, auto_create=False, auto_stop=True*)

Bases: object

__init__(*profile='default', cluster_dir=None, ipython_dir=None, auto_create=False, auto_stop=True*)

Create a class to manage an IPython cluster.

This class calls the **ipcluster** command with the right options to start an IPython cluster. Typically a cluster directory must be created (**ipcluster create**) and configured before using this class. Configuration is done by editing the configuration files in the top level of the cluster directory.

Parameters `profile` : str

The name of a cluster directory profile (default="default"). The cluster directory "cluster_<profile>" will be searched for in `os.getcwd()`, the `ipython_dir` and then in the directories listed in the **env:**`'IPCLUSTER_DIR_PATH'` environment variable.

cluster_dir : str

The full path to a cluster directory. This is useful if profiles are not being used.

ipython_dir : str

The location of the `ipython_dir` if different from the default. This is used if the cluster directory is being found by profile.

auto_create : bool

Automatically create the cluster directory if it doesn't exist. This will usually only make sense if using a local cluster (default=False).

auto_stop : bool

Automatically stop the cluster when this instance is garbage collected (default=True). This is useful if you want the cluster to live beyond your current process. There is also an instance attribute `auto_stop` to change this behavior.

auto_stop**get_ipcluster_logs()**

Get a dict of the ipcluster logs for this cluster.

get_ipcontroller_logs()

Get a dict of logs for the controller in this cluster.

get_ipengine_logs()

Get a dict of logs for all engines in this cluster.

get_logs()

Get a dict of all logs for this cluster.

get_logs_by_name (*name='ipcluster'*)

Get a dict of logs by process name (ipcluster, ipengine, etc.)

get_multiengine_client (*delay=0.20000000000000001, max_tries=9*)

Get the multiengine client for the running cluster.

This will try to attempt to the controller multiple times. If this fails altogether, try looking at the following: * Make sure the controller is starting properly by looking at its

log files.

- Make sure the controller is writing its FURL file in the location expected by the client.
- Make sure a firewall on the controller's host is not blocking the client from connecting.

Parameters **delay** : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

get_task_client (*delay=0.20000000000000001, max_tries=9*)

Get the task client for the running cluster.

This will try to attempt to the controller multiple times. If this fails altogether, try looking at the following: * Make sure the controller is starting properly by looking at its

log files.

- Make sure the controller is writing its FURL file in the location expected by the client.
- Make sure a firewall on the controller's host is not blocking the client from connecting.

Parameters **delay** : float

The initial delay between re-connection attempts. Susequent delays get longer according to `delay[i] = 1.5*delay[i-1]`.

max_tries : int

The max number of re-connection attempts.

location

print_ipcluster_logs ()

Print the ipcluster logs for this cluster to stdout.

print_ipcontroller_logs ()

Print the ipcontroller logs for this cluster to stdout.

print_ipengine_logs ()

Print the ipengine logs for this cluster to stdout.

print_logs ()

Print all the logs for this cluster to stdout.

running

start (*n=2*)

Start the IPython cluster with n engines.

Parameters **n** : int

The number of engine to start.

stop ()

Stop the IPython cluster if it is running.

ClusterStateError

```
class IPython.kernel.clientconnector.ClusterStateError
    Bases: exceptions.Exception

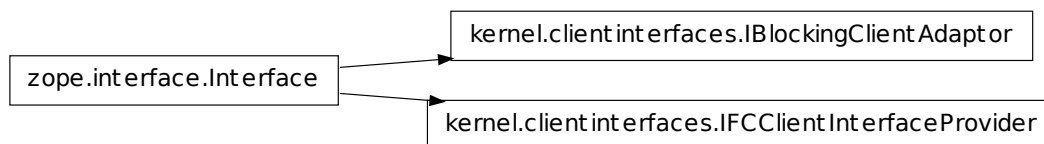
    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

8.37 kernel.clientinterfaces

8.37.1 Module: kernel.clientinterfaces

Inheritance diagram for IPython.kernel.clientinterfaces:



General client interfaces.

8.37.2 Classes

IBlockingClientAdaptor

```
class IPython.kernel.clientinterfaces.IBlockingClientAdaptor (name,
                                                             bases=(),
                                                             attrs=None,
                                                             __doc__=None,
                                                             __module__=None)

    Bases: zope.interface.Interface

    classmethod __init__ (name, bases=(), attrs=None, __doc__=None, __module__=None)

    classmethod changed (originally_changed)
        We, or something we depend on, have changed

    classmethod deferred ()
        Return a deferred class corresponding to the interface.
```

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

IFCClientInterfaceProvider

```
class IPython.kernel.clientinterfaces.IFCClientInterfaceProvider (name,
                                                                bases=(),
                                                                at-
                                                                trs=None,
                                                                __doc__=None,
                                                                __mod-
                                                                ule__=None)
```

Bases: `zope.interface.Interface`

classmethod `__init__` (*name*, *bases*=(), *attrs*=None, *__doc__*=None, *__module__*=None)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict*=True)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
```

```
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static **providedBy** ()
Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name*, *default=None*)

classmethod **queryTaggedValue** (*tag*, *default=None*)
Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag*, *value*)
Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj*, *errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

8.38 kernel.codeutil

8.38.1 Module: `kernel.codeutil`

Utilities to enable code objects to be pickled.

Any process that import this module will be able to pickle code objects. This includes the `func_code` attribute of any function. Once unpickled, new functions can be built using `new.function(code, globals())`. Eventually we need to automate all of this so that functions themselves can be pickled.

Reference: A. Tremols, P Cogolo, "Python Cookbook," p 302-305

8.38.2 Functions

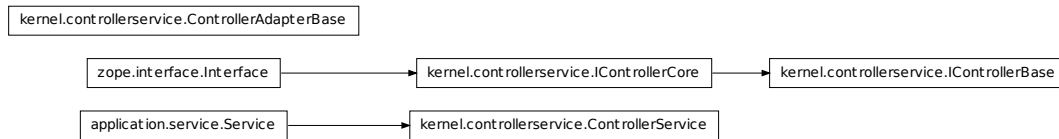
`IPython.kernel.codeutil.code_ctor (*args)`

`IPython.kernel.codeutil.reduce_code (co)`

8.39 kernel.controllerservice

8.39.1 Module: `kernel.controllerservice`

Inheritance diagram for `IPython.kernel.controllerservice`:



A Twisted Service for the IPython Controller.

The IPython Controller:

- Listens for Engines to connect and then manages access to those engines.
- Listens for clients and passes commands from client to the Engines.
- Exposes an asynchronous interfaces to the Engines which themselves can block.
- Acts as a gateway to the Engines.

The design of the controller is somewhat abstract to allow flexibility in how the controller is presented to clients. This idea is that there is a basic `ControllerService` class that allows engines to connect to it. But, this basic class has no client interfaces. To expose client interfaces developers provide an adapter that makes the `ControllerService` look like something. For example, one client interface might support task farming and another might support interactive usage. The important thing is that by using interfaces and adapters, a single controller can be accessed from multiple interfaces. Furthermore, by adapting various client interfaces to various network protocols, each client interface can be exposed to multiple network protocols. See `multiengine.py` for an example of how to adapt the `ControllerService` to a client interface.

8.39.2 Classes

`ControllerAdapterBase`

class `IPython.kernel.controllerservice.ControllerAdapterBase` (*controller*)

Bases: `object`

All Controller adapters should inherit from this class.

This class provides a wrapped version of the `IControllerBase` interface that can be used to easily create new custom controllers. Subclasses of this will provide a full implementation of `IControllerBase`.

This class doesn't implement any client notification mechanism. That is up to subclasses.

`__init__` (*controller*)

`on_n_engines_registered_do` (*n, f, *args, **kwargs*)

`on_register_engine_do` (*f, includeID, *args, **kwargs*)

`on_register_engine_do_not` (*f*)

`on_unregister_engine_do` (*f, includeID, *args, **kwargs*)

```

on_unregister_engine_do_not (f)
register_engine (remoteEngine, id=None, ip=None, port=None, pid=None)
unregister_engine (id)

```

ControllerService

```

class IPython.kernel.controllerservice.ControllerService (maxEngines=511,
                                                         saveIDs=False)

```

Bases: object, twisted.application.service.Service

A basic Controller represented as a Twisted Service.

This class doesn't implement any client notification mechanism. That is up to adapted subclasses.

```

__init__ (maxEngines=511, saveIDs=False)
disownServiceParent ()
on_n_engines_registered_do (n, f, *args, **kwargs)
on_register_engine_do (f, includeID, *args, **kwargs)
on_unregister_engine_do_not (f)
on_unregister_engine_do (f, includeID, *args, **kwargs)
on_unregister_engine_do_not (f)
privilegedStartService ()
register_engine (remoteEngine, id=None, ip=None, port=None, pid=None)
    Register new engine connection
setName (name)
setServiceParent (parent)
startService ()
stopService ()
unregister_engine (id)
    Unregister engine by id.

```

IControllerBase

```

class IPython.kernel.controllerservice.IControllerBase (name, bases=(),
                                                         attrs=None,
                                                         __doc__=None,
                                                         __module__=None)

```

Bases: IPython.kernel.controllerservice.IControllerCore

The basic controller interface.

```

classmethod __init__ (name, bases=(), attrs=None, __doc__=None, __module__=None)

```

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with ‘tag’.

classmethod `setTaggedValue (tag, value)`

Associates ‘value’ with ‘key’.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod weakref (*callback=None*)

IControllerCore

```
class IPython.kernel.controllerservice.IControllerCore(name, bases=(),
                                                       attrs=None,
                                                       __doc__=None,
                                                       __module__=None)
```

Bases: `zope.interface.Interface`

Basic methods any controller must have.

This is basically the aspect of the controller relevant to the engines and does not assume anything about how the engines will be presented to a client.

classmethod __init__ (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod changed (*originally_changed*)

We, or something we depend on, have changed

classmethod deferred ()

Return a deferred class corresponding to the interface.

classmethod direct (*name*)

classmethod extends (*interface, strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
```

```

0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod **namesAndDescriptions** (*all=False*)
Return attribute names and descriptions defined by interface.

static **providedBy** ()
Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name, default=None*)

classmethod **queryTaggedValue** (*tag, default=None*)
Returns the value associated with ‘tag’.

classmethod **setTaggedValue** (*tag, value*)
Associates ‘value’ with ‘key’.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

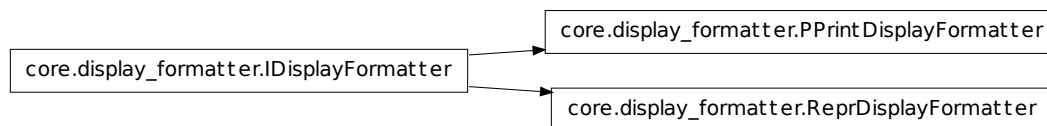
classmethod **validateInvariants** (*obj, errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

8.40 kernel.core.display_formatter

8.40.1 Module: `kernel.core.display_formatter`

Inheritance diagram for `IPython.kernel.core.display_formatter`:



Objects for replacing `sys.displayhook()`.

8.40.2 Classes

IDisplayFormatter

class `IPython.kernel.core.display_formatter.IDisplayFormatter`

Bases: `object`

Objects conforming to this interface will be responsible for formatting representations of objects that pass through `sys.displayhook()` during an interactive interpreter session.


```
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

PPrintDisplayFormatter

```
class IPython.kernel.core.display_formatter.PPrintDisplayFormatter
    Bases: IPython.kernel.core.display_formatter.IDisplayFormatter
    Return a pretty-printed string representation of an object.
```

```
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

ReprDisplayFormatter

```
class IPython.kernel.core.display_formatter.ReprDisplayFormatter
    Bases: IPython.kernel.core.display_formatter.IDisplayFormatter
    Return the repr() string representation of an object.
```

```
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

8.41 kernel.core.display_trap

8.41.1 Module: kernel.core.display_trap

Inheritance diagram for `IPython.kernel.core.display_trap`:

```
core.display_trap.DisplayTrap
```

Manager for replacing `sys.displayhook()`.

8.41.2 DisplayTrap

```
class IPython.kernel.core.display_trap.DisplayTrap (formatters=None,      call-
                                                    backs=None)
    Bases: object
    Object to trap and format objects passing through sys.displayhook().
```

This trap maintains two lists of callables: formatters and callbacks. The formatters take the *last* object that has gone through since the trap was set and returns a string representation. Callbacks are executed on *every* object that passes through the displayhook and does not return anything.

__init__ (*formatters=None, callbacks=None*)

add_to_message (*message*)

Add the formatted display of the objects to the message dictionary being returned from the interpreter to its listeners.

clear ()

Reset the stored object.

hook (*obj*)

This method actually implements the hook.

set ()

Set the hook.

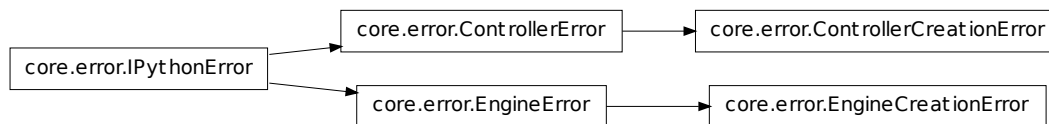
unset ()

Unset the hook.

8.42 kernel.core.error

8.42.1 Module: `kernel.core.error`

Inheritance diagram for `IPython.kernel.core.error`:



`error.py`

We declare here a class hierarchy for all exceptions produced by IPython, in cases where we don't just raise one from the standard library.

8.42.2 Classes

ControllerCreationError

class `IPython.kernel.core.error.ControllerCreationError`

Bases: `IPython.kernel.core.error.ControllerError`

```
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message
```

ControllerError

```
class IPython.kernel.core.error.ControllerError
    Bases: IPython.kernel.core.error.IPythonError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

EngineCreationError

```
class IPython.kernel.core.error.EngineCreationError
    Bases: IPython.kernel.core.error.EngineError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

EngineError

```
class IPython.kernel.core.error.EngineError
    Bases: IPython.kernel.core.error.IPythonError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

IPythonError

```
class IPython.kernel.core.error.IPythonError
    Bases: exceptions.Exception

    Base exception that all of our exceptions inherit from.


    This can be raised by code that doesn't have any more specific information.
```

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
args  
message
```

8.43 kernel.core.fd_redirector

8.43.1 Module: `kernel.core.fd_redirector`

Inheritance diagram for `IPython.kernel.core.fd_redirector`:



```
graph TD; A[core.fd_redirector.FDRedirector];
```

Stdout/stderr redirector, at the OS level, using file descriptors.

This also works under windows.

8.43.2 `FDRedirector`

class `IPython.kernel.core.fd_redirector.FDRedirector` (*fd=1*)

Bases: `object`

Class to redirect output (stdout or stderr) at the OS level using file descriptors.

__init__ (*fd=1*)
 fd is the file descriptor of the output you want to capture. It can be `STDOUT` or `STERR`.

flush ()
 Flush the captured output, similar to the flush method of any stream.

getvalue ()
 Return the output captured since the last getvalue, or the start of the redirection.

start ()
 Setup the redirection.

stop ()
 Unset the redirection and return the captured output.

8.44 kernel.core.file_like

8.44.1 Module: `kernel.core.file_like`

Inheritance diagram for `IPython.kernel.core.file_like`:



```

graph TD
    A[core.file_like.FileLike]
  
```

File like object that redirects its write calls to a given callback.

8.44.2 `FileLike`

class `IPython.kernel.core.file_like.FileLike` (*write_callback*)

Bases: `object`

`FileLike` object that redirects all write to a callback.

Only the write-related methods are implemented, as well as those required to read a `StringIO`.

__init__ (*write_callback*)

close ()

This method is there for compatibility with other file-like objects.

flush ()

This method is there for compatibility with other file-like objects.

getvalue ()

This method is there for compatibility with other file-like objects.

isatty ()

This method is there for compatibility with other file-like objects.

reset ()

This method is there for compatibility with other file-like objects.

truncate ()

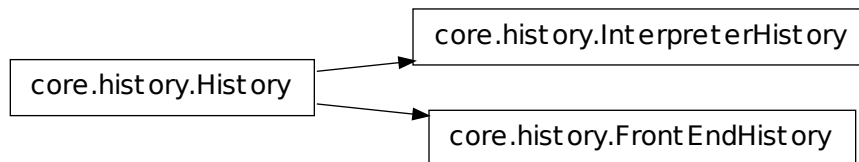
This method is there for compatibility with other file-like objects.

writelines (*lines*)

8.45 kernel.core.history

8.45.1 Module: `kernel.core.history`

Inheritance diagram for `IPython.kernel.core.history`:



Manage the input and output history of the interpreter and the frontend.

There are 2 different history objects, one that lives in the interpreter, and one that lives in the frontend. They are synced with a diff at each execution of a command, as the interpreter history is a real stack, its existing entries are not mutable.

8.45.2 Classes

FrontEndHistory

class `IPython.kernel.core.history.FrontEndHistory` (*input_cache=None*, *output_cache=None*)

Bases: `IPython.kernel.core.history.History`

An object managing the input and output history at the frontend. It is used as a local cache to reduce network latency problems and multiple users editing the same thing.

__init__ (*input_cache=None*, *output_cache=None*)

add_items (*item_list*)

Adds the given command list to the stack of executed commands.

get_history_item (*index*)

Returns the history string at index, where index is the distance from the end (positive).

History

class `IPython.kernel.core.history.History` (*input_cache=None*, *output_cache=None*)

Bases: `object`

An object managing the input and output history.

`__init__` (*input_cache=None, output_cache=None*)

`get_history_item` (*index*)

Returns the history string at index, where index is the distance from the end (positive).

InterpreterHistory

class `IPython.kernel.core.history.InterpreterHistory` (*input_cache=None, output_cache=None*)

Bases: `IPython.kernel.core.history.History`

An object managing the input and output history at the interpreter level.

`__init__` (*input_cache=None, output_cache=None*)

`get_history_item` (*index*)

Returns the history string at index, where index is the distance from the end (positive).

`get_input_after` (*index*)

Returns the list of the commands entered after index.

`get_input_cache` ()

`setup_namespace` (*namespace*)

Add the input and output caches into the interpreter's namespace with IPython-conventional names.

Parameters `namespace` : dict

`update_history` (*interpreter, python*)

Update the history objects that this object maintains and the interpreter's namespace.

Parameters `interpreter` : Interpreter

`python` : str

The real Python code that was translated and actually executed.

8.46 kernel.core.interpreter

8.46.1 Module: `kernel.core.interpreter`

Inheritance diagram for `IPython.kernel.core.interpreter`:

core.interpreter.Interpreter

core.interpreter.Not Defined

Central interpreter object for an IPython engine.

The interpreter is the object whose job is to process lines of user input and actually execute them in the user's namespace.

8.46.2 Classes

Interpreter

```
class IPython.kernel.core.interpreter.Interpreter(user_ns=None,  
                                                    global_ns=None,    trans-  
                                                    lator=None,   magic=None,  
                                                    display_formatters=None,  
                                                    traceback_formatters=None,  
                                                    output_trap=None,  
                                                    history=None,        mes-  
                                                    sage_cache=None,    file-  
                                                    name='<string>',    con-  
                                                    fig=None)
```

Bases: object

An interpreter object.

fixme: needs to negotiate available formatters with frontends.

Important: the interpeter should be built so that it exposes a method for each attribute/method of its sub-object. This way it can be replaced by a network adapter.

```
__init__(user_ns=None, global_ns=None, translator=None, magic=None, dis-  
         play_formatters=None, traceback_formatters=None, output_trap=None, his-  
         tory=None, message_cache=None, filename='<string>', config=None)
```

```
complete(line, text=None, pos=None)
```

Complete the given text.

Parameters

text [str] Text fragment to be completed on. Typically this is

```
error(text)
```

Pass an error message back to the shell.

Parameters `text` : str

Notes

This should only be called when `self.message` is set. In other words, when code is being executed.

execute (*commands*, *raiseException=True*)

Execute some IPython commands.

- 1.Translate them into Python.
- 2.Run them.
- 3.Trap stdout/stderr.
- 4.Trap `sys.displayhook()`.
- 5.Trap exceptions.
- 6.Return a message object.

Parameters `commands` : str

The raw commands that the user typed into the prompt.

Returns `message` : dict

The dictionary of responses. See the README.txt in this directory for an explanation of the format.

execute_block (*code*)

Execute a single block of code in the user namespace.

Return value: a flag indicating whether the code to be run completed successfully:

- 0: successful execution.
- 1: an error occurred.

execute_macro (*macro*)

Execute the value of a macro.

Parameters `macro` : Macro

execute_python (*python*)

Actually run the Python code in the namespace.

Parameters

python [str] Pure, exec'able Python code. Special IPython commands should have already been translated into pure Python.

feed_block (*source*, *filename*='<input>', *symbol*='single')

Compile some source in the interpreter.

One several things can happen:

- 1) The input is incorrect; `compile_command()` raised an exception (`SyntaxError` or `OverflowError`).
- 2) The input is incomplete, and more input is required; `compile_command()` returned `None`. Nothing happens.
- 3) The input is complete; `compile_command()` returned a code object. The code is executed by calling `self.runcode()` (which also handles run-time exceptions, except for `SystemExit`).

The return value is:

- True in case 2
- False in the other cases, unless an exception is raised, where

`None` is returned instead. This can be used by external callers to know whether to continue feeding input or not.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

format_traceback (*et*, *ev*, *tb*, *message*='')

Put a formatted version of the traceback into value and reraise.

When exceptions have to be sent over the network, the traceback needs to be put into the value of the exception in a nicely formatted way. The method takes the type, value and `tb` of an exception and puts a string representation of the `tb` into the value of the exception and reraises it.

Currently this method uses the `ultraTb` formatter from IPython trunk. Eventually it should simply use the traceback formatters in core that are loaded into `self.traceback_trap.formatters`.

generate_prompt (*is_continuation*)

Calculate and return a string with the prompt to display.

Parameters

is_continuation [bool] Whether the input line is continuing multiline input or not, so

that a proper continuation prompt can be computed.

get_command (*i*=None)

Gets the *i*th message in the `message_cache`.

This is implemented here for compatibility with the old `ipython1` shell I am not sure we need this though. I even seem to remember that we were going to get rid of it.

ipmagic (*arg_string*)

Call a magic function by name.

`ipmagic('name -opt foo bar')` is equivalent to typing at the `ipython` prompt:

In[1]: %name -opt foo bar

To call a magic without arguments, simply use `ipmagic('name')`.

This provides a proper Python function to call IPython's magics in any valid Python code you can type at the interpreter, including loops and compound statements. It is added by IPython to the Python builtin namespace upon initialization.

Parameters `arg_string` : str

A string containing the name of the magic function to call and any additional arguments to be passed to the magic.

Returns `something` : object

The return value of the actual object.

ipysystem (*command*)

Execute a command in a system shell while expanding variables in the current namespace.

Parameters `command` : str

pack_exception (*message, exc*)

pull (*keys*)

Get an item out of the namespace by key.

Parameters `key` : str

Returns `value` : object

Raises `TypeError` if the key is not a string. :

`NameError` if the object doesn't exist. :

pull_function (*keys*)

push (*ns*)

Put value into the namespace with name key.

Parameters `**kwds` :

push_function (*ns*)

reset ()

Reset the interpreter.

Currently this only resets the users variables in the namespace. In the future we might want to also reset the other stateful things like that the Interpreter has, like In, Out, etc.

set_traps ()

Set all of the output, display, and traceback traps.

setup_message ()

Return a message object.

This method prepares and returns a message dictionary. This dict contains the various fields that are used to transfer information about execution, results, tracebacks, etc, to clients (either in or out of process ones). Because of the need to work with possibly out of process clients, this dict MUST contain strictly pickle-safe values.

setup_namespace ()

Add things to the namespace.

split_commands (*python*)

Split multiple lines of code into discrete commands that can be executed singly.

Parameters *python* : str

Pure, exec'able Python code.

Returns *commands* : list of str

Separate commands that can be exec'ed independently.

unset_traps ()

Unset all of the output, display, and traceback traps.

var_expand (*template*)

Expand \$variables in the current namespace using Itpl.

Parameters *template* : str

NotDefined

class IPython.kernel.core.interpreter.**NotDefined**

Bases: object

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

8.46.3 Functions

IPython.kernel.core.interpreter.**default_display_formatters** ()

Return a list of default display formatters.

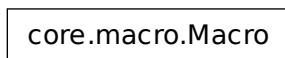
IPython.kernel.core.interpreter.**default_traceback_formatters** ()

Return a list of default traceback formatters.

8.47 kernel.core.macro

8.47.1 Module: kernel.core.macro

Inheritance diagram for IPython.kernel.core.macro:



Support for interactive macros in IPython

8.47.2 Macro

class IPython.kernel.core.magic.**Macro** (*data*)

Simple class to store the value of macros as strings.

This allows us to later exec them by checking when something is an instance of this class.

__init__ (*data*)

8.48 kernel.core.magic

8.48.1 Module: kernel.core.magic

Inheritance diagram for IPython.kernel.core.magic:



```
graph TD; A[core.magic.Magic];
```

8.48.2 Magic

class IPython.kernel.core.magic.**Magic** (*interpreter, config=None*)

Bases: object

An object that maintains magic functions.

__init__ (*interpreter, config=None*)

has_magic (*name*)

Return True if this object provides a given magic.

Parameters **name** : str

magic_env (*parameter_s=''*)

List environment variables.

magic_pwd (*parameter_s=''*)

Return the current working directory path.

object_find (*name*)

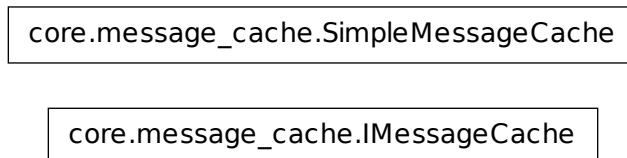
Find an object in the available namespaces.

fixme: this should probably be moved elsewhere. The interpreter?

8.49 kernel.core.message_cache

8.49.1 Module: `kernel.core.message_cache`

Inheritance diagram for `IPython.kernel.core.message_cache`:



Storage for the responses from the interpreter.

8.49.2 Classes

IMessageCache

class `IPython.kernel.core.message_cache.IMessageCache`

Bases: `object`

Storage for the response from the interpreter.

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`add_message(i, message)`

Add a message dictionary to the cache.

Parameters `i`: `int`

`message`: `dict`

`get_message(i=None)`

Get the message from the cache.

Parameters `i`: `int`, optional

The number of the message. If not provided, return the highest-numbered message.

Returns `message`: `dict`

Raises `IndexError` if the message does not exist in the cache. :

SimpleMessageCache

class IPython.kernel.core.message_cache.SimpleMessageCache

Bases: object

Simple dictionary-based, in-memory storage of the responses from the interpreter.

__init__()

add_message(i, message)

Add a message dictionary to the cache.

Parameters i: int

message: dict

get_message(i=None)

Get the message from the cache.

Parameters i: int, optional

The number of the message. If not provided, return the highest-numbered message.

Returns message: dict

Raises IndexError if the message does not exist in the cache. :

8.50 kernel.core.output_trap

8.50.1 Module: kernel.core.output_trap

Inheritance diagram for IPython.kernel.core.output_trap:

core.output_trap.OutputTrap

Trap stdout/stderr.

8.50.2 OutputTrap

class IPython.kernel.core.output_trap.OutputTrap(out=None, err=None)

Bases: object

Object which can trap text sent to stdout and stderr.

__init__(out=None, err=None)

add_to_message (*message*)

Add the text from stdout and stderr to the message from the interpreter to its listeners.

Parameters *message* : dict

clear ()

Clear out the buffers.

err_text

Return the text currently in the stderr buffer.

out_text

Return the text currently in the stdout buffer.

set ()

Set the hooks.

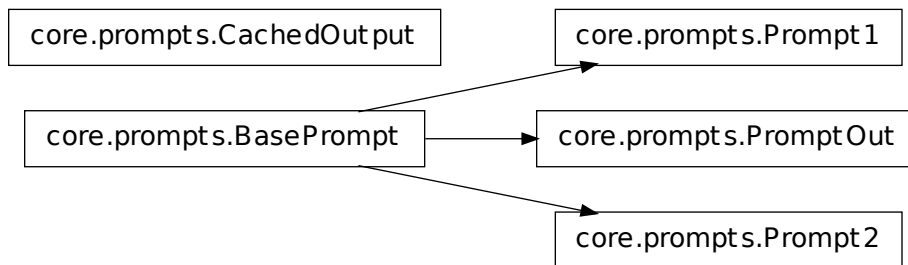
unset ()

Remove the hooks.

8.51 kernel.core.prompts

8.51.1 Module: `kernel.core.prompts`

Inheritance diagram for `IPython.kernel.core.prompts`:



Classes for handling input/output prompts.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.51.2 Classes

BasePrompt

class IPython.kernel.core.prompts.**BasePrompt** (*cache, sep, prompt, pad_left=False*)

Bases: object

Interactive prompt similar to Mathematica's.

__init__ (*cache, sep, prompt, pad_left=False*)

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_p_str ()

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write (*msg*)

CachedOutput

class IPython.kernel.core.prompts.**CachedOutput** (*shell, cache_size, Pprint, colors='NoColor', input_sep='n', output_sep='n', output_sep2=' ', ps1=None, ps2=None, ps_out=None, pad_left=True*)

Class for printing output from calculations while keeping a cache of results. It dynamically creates global variables prefixed with _ which contain these results.

Meant to be used as a sys.displayhook replacement, providing numbered prompts and cache services.

Initialize with initial and final values for cache counter (this defines the maximum size of the cache).

__init__ (*shell, cache_size, Pprint, colors='NoColor', input_sep='n', output_sep='n', output_sep2=' ', ps1=None, ps2=None, ps_out=None, pad_left=True*)

display (*arg*)

Default printer method, uses pprint.

Do ip.set_hook("result_display", my_displayhook) for custom result display, e.g. when your own objects need special formatting.

flush()

set_colors(colors)

Set the active color scheme and configure colors for the three prompt subsystems.

update(arg)

Prompt1

```
class IPython.kernel.core.prompts.Prompt1(cache, sep='n', prompt='In [#]: ',
                                           pad_left=True)
```

Bases: `IPython.kernel.core.prompts.BasePrompt`

Input interactive prompt similar to Mathematica's.

```
__init__(cache, sep='n', prompt='In [#]: ', pad_left=True)
```

auto_rewrite()

Print a string of the form '—>' which lines up with the previous input string. Useful for systems which re-write the user input when handling automatically special syntaxes.

cwd_filt(depth)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2(depth)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors()

set_p_str()

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write(msg)

Prompt2

```
class IPython.kernel.core.prompts.Prompt2(cache, prompt='.D.: ', pad_left=True)
```

Bases: `IPython.kernel.core.prompts.BasePrompt`

Interactive continuation prompt.

```
__init__(cache, prompt='.D.: ', pad_left=True)
```

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors ()

set_p_str ()

write (*msg*)

PromptOut

class IPython.kernel.core.prompts.**PromptOut** (*cache, sep=' ', prompt='Out[#]: ', pad_left=True*)

Bases: IPython.kernel.core.prompts.BasePrompt

Output interactive prompt similar to Mathematica's.

__init__ (*cache, sep=' ', prompt='Out[#]: ', pad_left=True*)

cwd_filt (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

cwd_filt2 (*depth*)

Return the last depth elements of the current working directory.

\$HOME is always replaced with '~'. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors ()

set_p_str ()

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write (*msg*)

8.51.3 Functions

`IPython.kernel.core.prompts.multiple_replace(dict, text)`

Replace in 'text' all occurrences of any key in the given dictionary by its corresponding value. Returns the new string.

`IPython.kernel.core.prompts.str_safe(arg)`

Convert to a string, without ever raising an exception.

If `str(arg)` fails, `<ERROR: ... >` is returned, where ... is the exception error message.

8.52 kernel.core.redirector_output_trap

8.52.1 Module: kernel.core.redirector_output_trap

Inheritance diagram for `IPython.kernel.core.redirector_output_trap`:



Trap stdout/stderr, including at the OS level. Calls a callback with the output each time Python tries to write to the stdout or stderr.

8.52.2 RedirectorOutputTrap

class `IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap(out_callback, err_callback)`

Bases: `IPython.kernel.core.output_trap.OutputTrap`

Object which can trap text sent to stdout and stderr.

__init__(*out_callback, err_callback*)

out_callback : callable called when there is output in the stdout *err_callback* : callable called when there is output in the stderr

add_to_message(*message*)

Add the text from stdout and stderr to the message from the interpreter to its listeners.

Parameters *message* : dict

clear()

Clear out the buffers.

err_text

Return the text currently in the stderr buffer.

on_err_write(*string*)

Callback called when there is some Python output on stderr.

on_out_write(*string*)

Callback called when there is some Python output on stdout.

out_text

Return the text currently in the stdout buffer.

set()

Set the hooks: set the redirectors and call the base class.

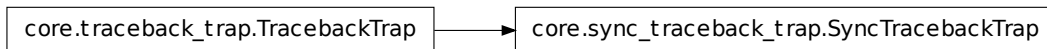
unset()

Remove the hooks: call the base class and stop the redirectors.

8.53 kernel.core.sync_traceback_trap

8.53.1 Module: kernel.core.sync_traceback_trap

Inheritance diagram for `IPython.kernel.core.sync_traceback_trap`:



Object to manage `sys.excepthook()`.

Synchronous version: prints errors when called.

8.53.2 SyncTracebackTrap

```

class IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap(
    sync_formatter=None,
    formatters=None,
    raiseException=True)
  
```

Bases: `IPython.kernel.core.traceback_trap.TracebackTrap`

`TracebackTrap` that displays immediately the traceback in addition to capturing it. Useful in frontends, as without this traceback trap, some tracebacks never get displayed.

__init__(*sync_formatter=None, formatters=None, raiseException=True*)

sync_formatter: Callable to display the traceback. *formatters*: A list of formatters to apply.

add_to_message (*message*)

Add the formatted display of the traceback to the message dictionary being returned from the interpreter to its listeners.

Parameters *message* : dict

clear ()

Remove the stored traceback.

hook (**args*)

This method actually implements the hook.

set ()

Set the hook.

unset ()

Unset the hook.

8.54 kernel.core.traceback_formatter

8.54.1 Module: `kernel.core.traceback_formatter`

Inheritance diagram for `IPython.kernel.core.traceback_formatter`:



Some formatter objects to extract traceback information by replacing `sys.excepthook()`.

8.54.2 Classes

`ITracebackFormatter`

class `IPython.kernel.core.traceback_formatter.ITracebackFormatter`

Bases: `object`

Objects conforming to this interface will format tracebacks into other objects.

__init__ ()

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`PlainTracebackFormatter`

class `IPython.kernel.core.traceback_formatter.PlainTracebackFormatter` (*limit=None*)

Bases: `IPython.kernel.core.traceback_formatter.ITracebackFormatter`

Return a string with the regular traceback information.

`__init__` (*limit=None*)

8.55 kernel.core.traceback_trap

8.55.1 Module: `kernel.core.traceback_trap`

Inheritance diagram for `IPython.kernel.core.traceback_trap`:

```
core.traceback_trap.TracebackTrap
```

Object to manage `sys.excepthook()`.

8.55.2 `TracebackTrap`

class `IPython.kernel.core.traceback_trap.TracebackTrap` (*formatters=None*)

Bases: `object`

Object to trap and format tracebacks.

`__init__` (*formatters=None*)

add_to_message (*message*)

Add the formatted display of the traceback to the message dictionary being returned from the interpreter to its listeners.

Parameters `message` : dict

clear ()

Remove the stored traceback.

hook (**args*)

This method actually implements the hook.

set ()

Set the hook.

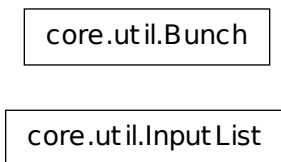
unset ()

Unset the hook.

8.56 kernel.core.util

8.56.1 Module: `kernel.core.util`

Inheritance diagram for `IPython.kernel.core.util`:



8.56.2 Classes

Bunch

```
class IPython.kernel.core.util.Bunch (*args, **kws)
    Bases: dict

    A dictionary that exposes its keys as attributes.

    __init__ (*args, **kws)

    clear
        D.clear() -> None. Remove all items from D.

    copy
        D.copy() -> a shallow copy of D

    static fromkeys ()
        dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.

    get
        D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

    has_key
        D.has_key(k) -> True if D has a key k, else False

    items
        D.items() -> list of D's (key, value) pairs, as 2-tuples

    iteritems
        D.iteritems() -> an iterator over the (key, value) items of D

    iterkeys
        D.iterkeys() -> an iterator over the keys of D
```


itervalues

D.itervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does: for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

InputList

class IPython.kernel.core.util.**InputList**

Bases: list

Class to store user input.

It's basically a list, but slices return a string instead of a list, thus allowing things like (assuming 'In' is an instance):

```
exec In[4:7]
```

or

```
exec In[5:9] + In[14] + In[21:25]
```

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

add (index, command)

Add a command to the list with the appropriate index.

If the index is greater than the current length of the list, empty strings are added in between.

append

L.append(object) – append object to end

count

L.count(value) -> integer – return number of occurrences of value

extend

L.extend(iterable) – extend list by appending elements from the iterable

index

L.index(value, [start, [stop]]) -> integer – return first index of value. Raises ValueError if the value is not present.

insert

L.insert(index, object) – insert object before index

pop

L.pop([index]) -> item – remove and return item at index (default last). Raises IndexError if list is empty or index is out of range.

remove

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse

L.reverse() – reverse *IN PLACE*

sort

L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

8.56.3 Functions

IPython.kernel.core.util.**esc_quotes** (strng)

Return the input string with single and double quotes escaped out.

IPython.kernel.core.util.**getoutputerror** (cmd, verbose=False, debug=False,
header='', split=False)

Executes a command and returns the output.

Parameters cmd : str

The command to execute.

verbose : bool

If True, print the command to be executed.

debug : bool

Only print, do not actually execute.

header : str

Header to print to screen prior to the executed command. No extra newlines are added.

split : bool

If True, return the output as a list split on newlines.

IPython.kernel.core.util.**make_quoted_expr** (s)

Return string s in appropriate quotes, using raw string if possible.

XXX - example removed because it caused encoding errors in documentation generation. We need a new example that doesn't contain invalid chars.

Note the use of raw string and padding at the end to allow trailing backslash.

```
IPython.kernel.core.util.system_shell(cmd, verbose=False, debug=False,  
                                         header='')
```

Execute a command in the system shell; always return None.

This returns None so it can be conveniently used in interactive loops without getting the return value (typically 0) printed many times.

Parameters **cmd** : str

The command to execute.

verbose : bool

If True, print the command to be executed.

debug : bool

Only print, do not actually execute.

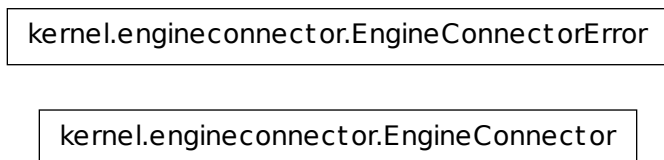
header : str

Header to print to screen prior to the executed command. No extra newlines are added.

8.57 kernel.engineconnector

8.57.1 Module: `kernel.engineconnector`

Inheritance diagram for `IPython.kernel.engineconnector`:



A class that manages the engines connection to the controller.

8.57.2 Classes

EngineConnector

class IPython.kernel.engineconnector.**EngineConnector** (*tub*)

Bases: object

Manage an engines connection to a controller.

This class takes a foolscap *Tub* and provides a *connect_to_controller* method that will use the *Tub* to connect to a controller and register the engine with the controller.

__init__ (*tub*)

connect_to_controller (**args, **kwargs*)

EngineConnectorError

class IPython.kernel.engineconnector.**EngineConnectorError**

Bases: exceptions.Exception

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

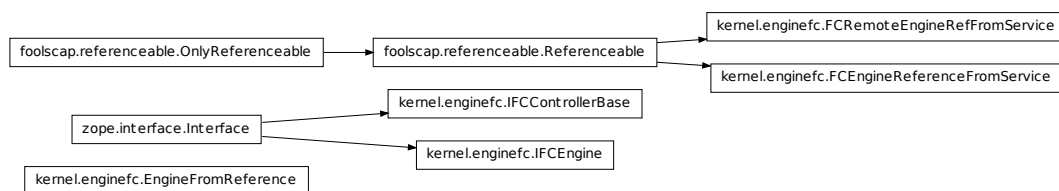
args

message

8.58 kernel.enginefc

8.58.1 Module: kernel.enginefc

Inheritance diagram for IPython.kernel.enginefc:



Expose the IPython EngineService using the Foolscap network protocol.

Foolscap is a high-performance and secure network protocol.

8.58.2 Classes

EngineFromReference

class IPython.kernel.enginefc.**EngineFromReference** (*reference*)

Bases: object

Adapt a *RemoteReference* to an *EngineBase* implementing object.

When an engine connects to a controller, it calls the *register_engine* method of the controller and passes the controller a *RemoteReference* to itself. This class is used to adapt this *RemoteReference* to an object that implements the full *EngineBase* interface.

See the documentation of *EngineBase* for details on the methods.

__init__ (*reference*)

callRemote (**args, **kwargs*)

checkReturnForFailure (*r*)

See if a returned value is a pickled Failure object.

To distinguish between general pickled objects and pickled Failures, the other side should prepend the string FAILURE: to any pickled Failure.

clear_properties ()

del_properties (*keys*)

execute (*lines*)

get_id ()

Return the Engines id.

get_properties (*keys=None*)

get_result (*i=None*)

has_properties (*keys*)

id

Return the Engines id.

keys ()

kill ()

killBack (*f*)

properties

pull (*keys*)

pull_function (*keys*)

pull_serialized (*keys*)

push (*namespace*)

push_function(*namespace*)

push_serialized(*namespace*)
Older version of pushSerialize.

reset()

set_id(*id*)
Set the Engines id.

set_properties(*properties*)

syncProperties(*r*)

FCEngineReferenceFromService

class IPython.kernel.enginefc.**FCEngineReferenceFromService**(*service*)
Bases: `foolscap.referenceable.Referenceable`, `object`

Adapt an *IEngineBase* to an *IFCEngine* implementer.

This exposes an *IEngineBase* to foolscap by adapting it to a *foolscap.Referenceable*.

See the documentation of the *IEngineBase* methods for more details.

__init__(*service*)

doRemoteCall(*methodname*, *args*, *kwargs*)

getInterface()

getInterfaceName()

processUniqueID()

remote_clear_properties()

remote_del_properties(*keys*)

remote_execute(*lines*)

remote_get_id()

remote_get_properties(*keys=None*)

remote_get_result(*i=None*)

remote_has_properties(*keys*)

remote_keys()

remote_kill()

remote_pull(*keys*)

remote_pull_function(*keys*)

remote_pull_serialized(*keys*)

remote_push(*pNamespace*)

```

remote_push_function(pNamespace)
remote_push_serialized(pNamespace)
remote_reset()
remote_set_id(id)
remote_set_properties(pNamespace)

```

FCRemoteEngineRefFromService

```

class IPython.kernel.enginefc.FCRemoteEngineRefFromService(service)
    Bases: foolscap.referenceable.Referenceable
    Adapt an IControllerBase to an IFCControllerBase.
    __init__(service)
    doRemoteCall(methodname, args, kwargs)
    getInterface()
    getInterfaceName()
    processUniqueID()
    remote_register_engine(engine_reference, id=None, pid=None, pproperties=None)

```

IFCControllerBase

```

class IPython.kernel.enginefc.IFCControllerBase(name, bases=(), attrs=None,
                                                __doc__=None, __module__=None)
    Bases: zope.interface.Interface

```

Interface that tells how an Engine sees a Controller.

In our architecture, the Controller listens for Engines to connect and register. This interface defines that registration method as it is exposed over the Foolscape network protocol

```

classmethod __init__(name, bases=(), attrs=None, __doc__=None, __module__=None)

```

```

classmethod changed(originally_changed)
    We, or something we depend on, have changed

```

```

classmethod deferred()
    Return a deferred class corresponding to the interface.

```

```

classmethod direct(name)

```

```

classmethod extends(interface, strict=True)
    Does the specification extend the given interface?

```

```

    Test whether an interface in the specification extends the given interface

```

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod interfaces ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod isEqualOrExtendedBy (other)

Same interface or extends?

static isOrExtends ()

Test whether a specification is or extends another

classmethod names (all=False)

Return the attribute names defined by the interface.

classmethod namesAndDescriptions (all=False)

Return attribute names and descriptions defined by interface.

static providedBy ()

Test whether an interface is implemented by the specification

classmethod queryDescriptionFor (name, default=None)**classmethod queryTaggedValue (tag, default=None)**

Returns the value associated with 'tag'.

classmethod setTaggedValue (tag, value)

Associates 'value' with 'key'.

classmethod subscribe (dependent)**classmethod unsubscribe (dependent)****classmethod validateInvariants (obj, errors=None)**

validate object to defined invariants.

classmethod weakref (callback=None)**IFCEngine**

```
class IPython.kernel.enginefc.IFCEngine (name, bases=(), attrs=None,
                                         __doc__=None, __module__=None)
```

Bases: `zope.interface.Interface`

An interface that exposes an EngineService over Foolsap.

The methods in this interface are similar to those from IEngine, but their arguments and return values slightly different to reflect that FC cannot send arbitrary objects. We handle this by pickling/unpickling that the two endpoints.

If a remote or local exception is raised, the appropriate Failure will be returned instead.

classmethod `__init__` (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases ()`

classmethod `getDescriptionFor (name)`

Return the attribute description for the given name.

classmethod `getDoc ()`

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)
Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

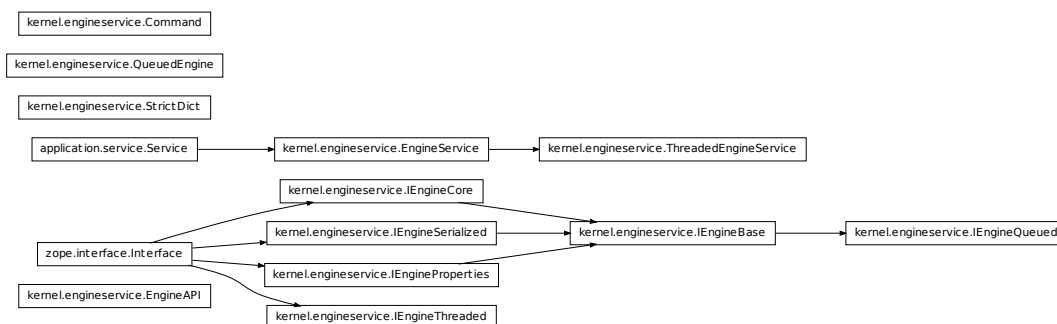
classmethod **validateInvariants** (*obj, errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

8.59 kernel.engineservice

8.59.1 Module: `kernel.engineservice`

Inheritance diagram for `IPython.kernel.engineservice`:



A Twisted Service Representation of the IPython core.

The IPython Core exposed to the network is called the Engine. Its representation in Twisted is the Engine-Service. Interfaces and adapters are used to abstract out the details of the actual network protocol used. The EngineService is an Engine that knows nothing about the actual protocol used.

The EngineService is exposed with various network protocols in modules like:

`enginepb.py` `enginevanilla.py`

As of 12/12/06 the classes in this module have been simplified greatly. It was felt that we had over-engineered things. To improve the maintainability of the code we have taken out the `ICompleteEngine` interface and the `completeEngine` method that automatically added methods to engines.

8.59.2 Classes

Command

class IPython.kernel.engineservice.**Command** (*remoteMethod*, *args, **kwargs)

Bases: object

A command object that encapsulates queued commands.

This class basically keeps track of a command that has been queued in a QueuedEngine. It manages the deferreds and hold the method to be called and the arguments to that method.

__init__ (*remoteMethod*, *args, **kwargs)

Build a new Command object.

handleError (*reason*)

When an error has occurred, relay it to self.deferred.

handleResult (*result*)

When the result is ready, relay it to self.deferred.

setDeferred (*d*)

Sets the deferred attribute of the Command.

EngineAPI

class IPython.kernel.engineservice.**EngineAPI** (*id*)

Bases: object

This is the object through which the user can edit the *properties* attribute of an Engine. The Engine Properties object copies all object in and out of itself. See the EngineProperties object for details.

__init__ (*id*)

EngineService

class IPython.kernel.engineservice.**EngineService** (*shellClass*=<class 'IPython.kernel.core.interpreter.Interpreter'>, *mpi*=None)

Bases: object, twisted.application.service.Service

Adapt a IPython shell into a IEngine implementing Twisted Service.

__init__ (*shellClass*=<class 'IPython.kernel.core.interpreter.Interpreter'>, *mpi*=None)

Create an EngineService.

shellClass: something that implements IInterpreter or core.I mpi: an mpi module that has rank and size attributes

addIDToResult (*result*)

clear_properties ()

del_properties (*keys*)

disownServiceParent ()

execute (*lines*)

executeAndRaise (*msg, callable, *args, **kwargs*)

Call a method of self.shell and wrap any exception.

get_properties (*keys=None*)

get_result (*i=None*)

has_properties (*keys*)

id

keys ()

Return a list of variables names in the users top level namespace.

This used to return a dict of all the keys/repr(values) in the user's namespace. This was too much info for the ControllerService to handle so it is now just a list of keys.

kill ()

privilegedStartService ()

pull (*keys*)

pull_function (*keys*)

pull_serialized (*keys*)

push (*namespace*)

push_function (*namespace*)

push_serialized (*sNamespace*)

reset ()

setName (*name*)

setServiceParent (*parent*)

set_properties (*properties*)

startService ()

stopService ()

IEngineBase

```
class IPython.kernel.engineservice.IEngineBase (name, bases=(), attrs=None,
                                                __doc__=None, __module__=None)
    Bases: IPython.kernel.engineservice.IEngineCore,
```

```
IPython.kernel.engineservice.IEngineSerialized,
IPython.kernel.engineservice.IEngineProperties
```

The basic engine interface that EngineService will implement.

This exists so it is easy to specify adapters that adapt to and from the API that the basic EngineService implements.

classmethod `__init__` (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases ()`

classmethod `getDescriptionFor (name)`

Return the attribute description for the given name.

classmethod `getDoc ()`

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)
 Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj, errors=None*)
 validate object to defined invariants.

classmethod **weakref** (*callback=None*)

IEngineCore

class IPython.kernel.engineservice.**IEngineCore** (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

Bases: `zope.interface.Interface`

The minimal required interface for the IPython Engine.

This interface provides a formal specification of the IPython core. All these methods should return deferreds regardless of what side of a network connection they are on.

In general, this class simply wraps a shell class and wraps its return values as Deferred objects. If the underlying shell class method raises an exception, this class should convert it to a `twisted.failure.Failure` that will be propagated along the Deferred's errback chain.

In addition, Failures are aggressive. By this, we mean that if a method is performing multiple actions (like pulling multiple object) if any single one fails, the entire method will fail with that Failure. It is all or nothing.

classmethod **__init__** (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod **changed** (*originally_changed*)
 We, or something we depend on, have changed

classmethod **deferred** ()
 Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface, strict=True*)
 Does the specification extend the given interface?
 Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
```

```
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
```

```

...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with ‘tag’.

classmethod `setTaggedValue` (*tag, value*)

Associates ‘value’ with ‘key’.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

IEngineProperties

```

class IPython.kernel.engineservice.IEngineProperties (name,          bases=(),
                                                    attrs=None,
                                                    __doc__=None, __mod-
                                                    ule__=None)

```

Bases: `zope.interface.Interface`

Methods for access to the properties object of an Engine

classmethod `__init__` (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod **deferred** ()

Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod **get** (*name*, *default=None*)

Query for an attribute description

classmethod **getBases** ()

classmethod **getDescriptionFor** (*name*)

Return the attribute description for the given name.

classmethod **getDoc** ()

Returns the documentation for the object.

classmethod **getName** ()

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

IEngineQueued

```
class IPython.kernel.engineservice.IEngineQueued(name, bases=(), attrs=None,
                                                  __doc__=None, __module__=None,
                                                  __le__=None)
```

Bases: IPython.kernel.engineservice.IEngineBase

Interface for adding a queue to an IEngineBase.

This interface extends the IEngineBase interface to add methods for managing the engine's queue. The implicit details of this interface are that the execution of all methods declared in IEngineBase should appropriately be put through a queue before execution.

All methods should return deferreds.

```
classmethod __init__(name, bases=(), attrs=None, __doc__=None, __module__=None)
```

```
classmethod changed(originally_changed)
```

We, or something we depend on, have changed

```
classmethod deferred()
```

Return a deferred class corresponding to the interface.

```
classmethod direct(name)
```

```
classmethod extends(interface, strict=True)
```

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
```

```

>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod **namesAndDescriptions** (*all=False*)
Return attribute names and descriptions defined by interface.

static **providedBy** ()
Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name, default=None*)

classmethod **queryTaggedValue** (*tag, default=None*)
Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)
Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj, errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

IEngineSerialized

```
class IPython.kernel.engineservice.IEngineSerialized(name, bases=(),
                                                    attrs=None,
                                                    __doc__=None, __module__=None)
```

Bases: `zope.interface.Interface`

Push/Pull methods that take Serialized objects.

All methods should return deferreds.

classmethod **__init__** (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod **changed** (*originally_changed*)

We, or something we depend on, have changed

classmethod **deferred** ()

Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface, strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...

```



```

>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod **isEqualOrExtendedBy** (*other*)

Same interface or extends?

static **isOrExtends** ()

Test whether a specification is or extends another

classmethod **names** (*all=False*)

Return the attribute names defined by the interface.

classmethod **namesAndDescriptions** (*all=False*)

Return attribute names and descriptions defined by interface.

static **providedBy** ()

Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name, default=None*)

classmethod **queryTaggedValue** (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)

Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj, errors=None*)

validate object to defined invariants.

classmethod **weakref** (*callback=None*)

IEngineThreaded

```
class IPython.kernel.engineservice.IEngineThreaded (name, bases=(), at-
                                                    trs=None, __doc__=None,
                                                    __module__=None)
```

Bases: `zope.interface.Interface`

A place holder for threaded commands.

All methods should return deferreds.

classmethod **__init__** (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with ‘tag’.

classmethod `setTaggedValue (tag, value)`

Associates ‘value’ with ‘key’.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod weakref (*callback=None*)

QueuedEngine

class IPython.kernel.engineservice.**QueuedEngine** (*engine*)

Bases: object

Adapt an IEngineBase to an IEngineQueued by wrapping it.

The resulting object will implement IEngineQueued which extends IEngineCore which extends (IEngineBase, IEngineSerialized).

This seems like the best way of handling it, but I am not sure. The other option is to have the various base interfaces be used like mix-in interfaces. The problem I have with this is adaptation is more difficult and complicated because there can be multiple original and final Interfaces.

__init__ (*engine*)

Create a QueuedEngine object from an engine

engine: An implementor of IEngineCore and IEngineSerialized keepUpToDate: whether to update the remote status when the

queue is empty. Defaults to False.

abortCommand (*reason*)

Abort current command.

This eats the Failure but first passes it onto the Deferred that the user has.

It also clear out the queue so subsequence commands don't run.

clear_properties (*this, *args, **kwargs*)

clear_queue (*msg=''*)

Clear the queue, but doesn't cancel the currently running command.

del_properties (*this, *args, **kwargs*)

execute (*this, *args, **kwargs*)

finishCommand (*result*)

Finish current command.

get_properties (*this, *args, **kwargs*)

get_result (*i=None*)

has_properties (*this, *args, **kwargs*)

keys (*this, *args, **kwargs*)

kill ()

properties

pull (*this, *args, **kwargs*)

pull_function (*this, *args, **kwargs*)

```
pull_serialized(this, *args, **kwargs)
push(this, *args, **kwargs)
push_function(this, *args, **kwargs)
push_serialized(this, *args, **kwargs)
queue_status()
register_failure_observer(obs)
reset()
runCurrentCommand()
    Run current command.
saveResult(result)
    Put the result in the history.
set_properties(this, *args, **kwargs)
submitCommand(cmd)
    Submit command to queue.
unregister_failure_observer(obs)
```

StrictDict

```
class IPython.kernel.engineservice.StrictDict(*args, **kwargs)
    Bases: dict
```

This is a strict copying dictionary for use as the interface to the properties of an Engine.

Important This object copies the values you set to it, and returns copies to you when you request them. The only way to change properties is explicitly through the setitem and getitem of the dictionary interface.

Example:

```
>>> e = get_engine(id)
>>> L = [1,2,3]
>>> e.properties['L'] = L
>>> L == e.properties['L']
True
>>> L.append(99)
>>> L == e.properties['L']
False
```

Note that getitem copies, so calls to methods of objects do not affect the properties, as seen here:

```
>>> e.properties[1] = range(2)
>>> print e.properties[1]
[0, 1]
>>> e.properties[1].append(2)
```

```
>>> print e.properties[1]
[0, 1]
```

__init__ (*args, **kwargs)

clear ()

copy

D.copy() -> a shallow copy of D

static fromkeys ()

dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.

get

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

has_key

D.has_key(k) -> True if D has a key k, else False

items

D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems

D.iteritems() -> an iterator over the (key, value) items of D

iterkeys

D.iterkeys() -> an iterator over the keys of D

intervalues

D.intervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop (key)

popitem ()

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

subDict (*keys)

update (dikt)

values

D.values() -> list of D's values

ThreadedEngineService

```
class IPython.kernel.engineservice.ThreadedEngineService (shellClass=<class
                                                             'IPython.kernel.core.interpreter.Interpreter'
                                                             mpi=None)

Bases: IPython.kernel.engineservice.EngineService
```

An EngineService subclass that defers execute commands to a separate thread.

ThreadedEngineService uses twisted.internet.threads.deferToThread to defer execute requests to a separate thread. GUI frontends may want to use ThreadedEngineService as the engine in an IPython.frontend.frontendbase.FrontEndBase subclass to prevent block execution from blocking the GUI thread.

```
__init__ (shellClass=<class 'IPython.kernel.core.interpreter.Interpreter'>, mpi=None)  
addIDToResult (result)  
clear_properties ()  
del_properties (keys)  
disownServiceParent ()  
execute (lines)  
executeAndRaise (msg, callable, *args, **kwargs)  
    Call a method of self.shell and wrap any exception.  
get_properties (keys=None)  
get_result (i=None)  
has_properties (keys)  
id  
keys ()  
    Return a list of variables names in the users top level namespace.  
  
    This used to return a dict of all the keys/repr(values) in the user's namespace. This was too much  
    info for the ControllerService to handle so it is now just a list of keys.  
  
kill ()  
privilegedStartService ()  
pull (keys)  
pull_function (keys)  
pull_serialized (keys)  
push (namespace)  
push_function (namespace)  
push_serialized (sNamespace)  
reset ()  
setName (name)  
setServiceParent (parent)  
set_properties (properties)  
startService ()
```


stopService()

wrapped_execute (*msg, lines*)

Wrap self.shell.execute to add extra information to tracebacks

8.59.3 Functions

IPython.kernel.engineservice.**drop_engine** (*id*)

remove an engine

IPython.kernel.engineservice.**get_engine** (*id*)

Get the Engine API object, whcih currently just provides the properties object, by ID

IPython.kernel.engineservice.**queue** (*methodToQueue*)

8.60 kernel.error

8.60.1 Module: kernel.error

Inheritance diagram for IPython.kernel.error:



Classes and functions for kernel related errors and exceptions.

8.60.2 Classes

AbortedPendingDeferredError

```
class IPython.kernel.error.AbortedPendingDeferredError
    Bases: IPython.kernel.error.KernelError
```

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
args  
message
```

ClientError

```
class IPython.kernel.error.ClientError  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

CompositeError

```
class IPython.kernel.error.CompositeError (message, elist)  
    Bases: IPython.kernel.error.KernelError  
  
    __init__ (message, elist)  
  
    args  
  
    message  
  
    print_tracebacks (excid=None)  
  
    raise_exception (excid=0)
```

ConnectionError

```
class IPython.kernel.error.ConnectionError  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

FileTimeoutError

```
class IPython.kernel.error.FileTimeoutError  
    Bases: IPython.kernel.error.KernelError
```

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
args  
message
```

IdInUse

```
class IPython.kernel.error.IdInUse  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

InvalidClientID

```
class IPython.kernel.error.InvalidClientID  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

InvalidDeferredID

```
class IPython.kernel.error.InvalidDeferredID  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

InvalidEngineID

```
class IPython.kernel.error.InvalidEngineID  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args
```

message

InvalidProperty

class IPython.kernel.error.InvalidProperty

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

KernelError

class IPython.kernel.error.KernelError

Bases: IPython.kernel.core.error.IPythonError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

MessageSizeError

class IPython.kernel.error.MessageSizeError

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

MissingBlockArgument

class IPython.kernel.error.MissingBlockArgument

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

NoEnginesRegistered

```
class IPython.kernel.error.NoEnginesRegistered
    Bases: IPython.kernel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

NotAPendingResult

```
class IPython.kernel.error.NotAPendingResult
    Bases: IPython.kernel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

NotDefined

```
class IPython.kernel.error.NotDefined(name)
    Bases: IPython.kernel.error.KernelError

    __init__(name)

    args
    message
```

PBMessageSizeError

```
class IPython.kernel.error.PBMessageSizeError
    Bases: IPython.kernel.error.MessageSizeError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

ProtocolError

```
class IPython.kernel.error.ProtocolError
    Bases: IPython.kernel.error.KernelError
```

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
args  
message
```

QueueCleared

```
class IPython.kernel.error.QueueCleared  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

ResultAlreadyRetrieved

```
class IPython.kernel.error.ResultAlreadyRetrieved  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

ResultNotCompleted

```
class IPython.kernel.error.ResultNotCompleted  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

SecurityError

```
class IPython.kernel.error.SecurityError  
    Bases: IPython.kernel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args
```

message

SerializationError

class IPython.kernel.error.**SerializationError**

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

StopLocalExecution

class IPython.kernel.error.**StopLocalExecution**

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

TaskAborted

class IPython.kernel.error.**TaskAborted**

Bases: IPython.kernel.error.KernelError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

TaskRejectError

class IPython.kernel.error.**TaskRejectError**

Bases: IPython.kernel.error.KernelError

Exception to raise when a task should be rejected by an engine.

This exception can be used to allow a task running on an engine to test if the engine (or the user's namespace on the engine) has the needed task dependencies. If not, the task should raise this exception. For the task to be retried on another engine, the task should be created with the *retries* argument > 1.

The advantage of this approach over our older properties system is that tasks have full access to the user's namespace on the engines and the properties don't have to be managed or tested by the controller.

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

args

message

TaskTimeout

```
class IPython.kernel.error.TaskTimeout
```

Bases: `IPython.kernel.error.KernelError`

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

args

message

UnpickleableException

```
class IPython.kernel.error.UnpickleableException
```

Bases: `IPython.kernel.error.KernelError`

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

args

message

8.60.3 Function

`IPython.kernel.error.collect_exceptions` (*rlist, method*)

8.61 kernel.ipclusterapp

8.61.1 Module: kernel.ipclusterapp

Inheritance diagram for `IPython.kernel.ipclusterapp`:



The ipcluster application.

8.61.2 Classes

IPClusterApp

```
class IPython.kernel.ipclusterapp.IPClusterApp(argv=None)
    Bases: IPython.kernel.clusterdir.ApplicationWithClusterDir

    __init__(argv=None)

    attempt(func)

    command_line_loader
        alias of IPClusterAppConfigLoader

    construct()

    crash_handler_class
        alias of ClusterDirCrashHandler

    create_command_line_config()
        Create and return a command line config loader.

    create_crash_handler()
        Create a crash handler, typically setting sys.excepthook to it.

    create_default_config()

    exit(exit_status=0)

    find_config_file_name()
        Find the config file name for this application.

    find_config_file_paths()

    find_ipython_dir()
        Set the IPython directory.

        This sets self.ipython_dir, but the actual value that is passed to the application is kept
        in either self.default_config or self.command_line_config. This also adds
        self.ipython_dir to sys.path so config files there can be referenced by other config
        files.

    find_resources()

    finish_cluster_dir()
```

get_pid_from_file()

Get the pid from the pid file.

If the pid file doesn't exist a `PIDFileError` is raised.

init_logger()

initialize()

Initialize the application.

Loads all configuration information and sets all application state, but does not start any relevant processing (typically some kind of event loop).

Once this method has been called, the application is flagged as initialized and the method becomes a no-op.

list_cluster_dirs()

load_command_line_config()

Load the command line config.

load_file_config()

Load the config file.

This tries to load the config file from disk. If successful, the `CONFIG_FILE` config variable is set to the resolved config file location. If not successful, an empty config is used.

log_command_line_config()

log_default_config()

log_err(f)

log_file_config()

log_level

log_master_config()

merge_configs()

Merge the default, command line and file config objects.

post_construct()

Do actions after construct, but before starting the app.

post_load_command_line_config()

Do actions just after loading the command line config.

post_load_file_config()

Do actions after the config file is loaded.

pre_construct()

pre_load_command_line_config()

Do actions just before loading the command line config.

pre_load_file_config()

Do actions before the config file is loaded.

remove_pid_file()

Remove the pid file.

This should be called at shutdown by registering a callback with `reactor.addSystemEventTrigger()`. This needs to return `None`.

set_command_line_config_log_level()

set_default_config_log_level()

set_file_config_log_level()

sigint_handler (*signum, frame*)

start()

Start the application.

start_app()

Start the application, depending on what subcommand is used.

start_app_start()

Start the app for the start subcommand.

start_app_stop()

Start the app for the stop subcommand.

start_controller (*r=None*)

start_engines (*r=None*)

start_launchers()

start_logging()

startup_message (*r=None*)

stop_controller (*r=None*)

stop_engines (*r=None*)

stop_launchers (*r=None*)

to_work_dir()

write_pid_file (*overwrite=False*)

Create a .pid file in the pid_dir with my pid.

This must be called after `pre_construct`, which sets `self.pid_dir`. This raises `PIDFileError` if the pid file exists already.

IPClusterAppConfigLoader

```
class IPython.kernel.ipclusterapp.IPClusterAppConfigLoader (argv=None,
                                                            *parser_args,
                                                            **parser_kw)
```

Bases: `IPython.kernel.clusterdir.ClusterDirConfigLoader`

`__init__(argv=None, *parser_args, **parser_kw)`

Create a config loader for use with argparse.

Parameters `argv` : optional, list

If given, used to read command-line arguments from, otherwise `sys.argv[1:]` is used.

parser_args : tuple

A tuple of positional arguments that will be passed to the constructor of `argparse.ArgumentParser`.

parser_kw : dict

A tuple of keyword arguments that will be passed to the constructor of `argparse.ArgumentParser`.

`clear()`

`get_extra_args()`

`load_config(args=None)`

Parse command line arguments and return as a Struct.

Parameters `args` : optional, list

If given, a list with the structure of `sys.argv[1:]` to parse arguments from. If not given, the instance's `self.argv` attribute (given at construction time) is used.

8.61.3 Function

`IPython.kernel.ipclusterapp.launch_new_instance()`

Create and run the IPython cluster.

8.62 kernel.ipengineapp

8.62.1 Module: `kernel.ipengineapp`

Inheritance diagram for `IPython.kernel.ipengineapp`:



The IPython controller application

8.62.2 Classes

IPEngineApp

```
class IPython.kernel.ipengineapp.IPEngineApp (argv=None)
    Bases: IPython.kernel.clusterdir.ApplicationWithClusterDir

    __init__ (argv=None)

    attempt (func)

    call_connect ()

    command_line_loader
        alias of IPEngineAppConfigLoader

    construct ()

    crash_handler_class
        alias of ClusterDirCrashHandler

    create_command_line_config ()
        Create and return a command line config loader.

    create_crash_handler ()
        Create a crash handler, typically setting sys.excepthook to it.

    create_default_config ()

    exec_lines ()

    exit (exit_status=0)

    find_config_file_name ()
        Find the config file name for this application.

    find_config_file_paths ()

    find_cont_furl_file ()
        Set the furl file.

        Here we don't try to actually see if it exists for is valid as that is hadled by the connection logic.

    find_ipython_dir ()
        Set the IPython directory.

        This sets self.ipython_dir, but the actual value that is passed to the application is kept
        in either self.default_config or self.command_line_config. This also adds
        self.ipython_dir to sys.path so config files there can be referenced by other config
        files.

    find_resources ()
        This resolves the cluster directory.

        This tries to find the cluster directory and if successful, it will have done: * Sets
        self.cluster_dir_obj to the ClusterDir object for
```

the application.

- Sets `self.cluster_dir` attribute of the application and config objects.

The algorithm used for this is as follows: 1. Try `Global.cluster_dir`. 2. Try using `Global.profile`. 3. If both of these fail and `self.auto_create_cluster_dir` is

True, then create the new cluster dir in the IPython directory.

4.If all fails, then raise `ClusterDirError`.

`finish_cluster_dir()`

`get_pid_from_file()`

Get the pid from the pid file.

If the pid file doesn't exist a `PIDFileError` is raised.

`init_logger()`

`initialize()`

Initialize the application.

Loads all configuration information and sets all application state, but does not start any relevant processing (typically some kind of event loop).

Once this method has been called, the application is flagged as initialized and the method becomes a no-op.

`load_command_line_config()`

Load the command line config.

`load_file_config()`

Load the config file.

This tries to load the config file from disk. If successful, the `CONFIG_FILE` config variable is set to the resolved config file location. If not successful, an empty config is used.

`log_command_line_config()`

`log_default_config()`

`log_file_config()`

`log_level`

`log_master_config()`

`merge_configs()`

Merge the default, command line and file config objects.

`post_construct()`

Do actions after construct, but before starting the app.

`post_load_command_line_config()`

post_load_file_config()

Do actions after the config file is loaded.

pre_construct()

pre_load_command_line_config()

Do actions just before loading the command line config.

pre_load_file_config()

Do actions before the config file is loaded.

remove_pid_file()

Remove the pid file.

This should be called at shutdown by registering a callback with `reactor.addSystemEventTrigger()`. This needs to return `None`.

set_command_line_config_log_level()

set_default_config_log_level()

set_file_config_log_level()

start()

Start the application.

start_app()

start_logging()

start_mpi()

to_work_dir()

write_pid_file(overwrite=False)

Create a .pid file in the pid_dir with my pid.

This must be called after `pre_construct`, which sets `self.pid_dir`. This raises `PIDFileError` if the pid file exists already.

IPythonEngineAppConfigLoader

```
class IPython.kernel.ipengineapp.IPythonEngineAppConfigLoader (argv=None,  
                                                             *parser_args,  
                                                             **parser_kw)
```

Bases: `IPython.kernel.clusterdir.ClusterDirConfigLoader`

```
__init__(argv=None, *parser_args, **parser_kw)
```

Create a config loader for use with `argparse`.

Parameters `argv` : optional, list

If given, used to read command-line arguments from, otherwise `sys.argv[1:]` is used.

`parser_args` : tuple

A tuple of positional arguments that will be passed to the constructor of `argparse.ArgumentParser`.

parser_kw : dict

A tuple of keyword arguments that will be passed to the constructor of `argparse.ArgumentParser`.

clear()

get_extra_args()

load_config (*args=None*)

Parse command line arguments and return as a Struct.

Parameters **args** : optional, list

If given, a list with the structure of `sys.argv[1:]` to parse arguments from. If not given, the instance's `self.argv` attribute (given at construction time) is used.

SimpleStruct

8.62.3 Function

`IPython.kernel.ipengineapp.launch_new_instance()`

Create and run the IPython controller

8.63 kernel.map

8.63.1 Module: `kernel.map`

Inheritance diagram for `IPython.kernel.map`:



Classes used in scattering and gathering sequences.

Scattering consists of partitioning a sequence and sending the various pieces to individual nodes in a cluster.

8.63.2 Classes

Map

class `IPython.kernel.map.Map`
A class for partitioning a sequence using a map.

concatenate (*listOfPartitions*)

getPartition (*seq, p, q*)
Returns the pth partition of q partitions of seq.

joinPartitions (*listOfPartitions*)

RoundRobinMap

class `IPython.kernel.map.RoundRobinMap`
Bases: `IPython.kernel.map.Map`
Partitions a sequence in a round robin fashion.
This currently does not work!

concatenate (*listOfPartitions*)

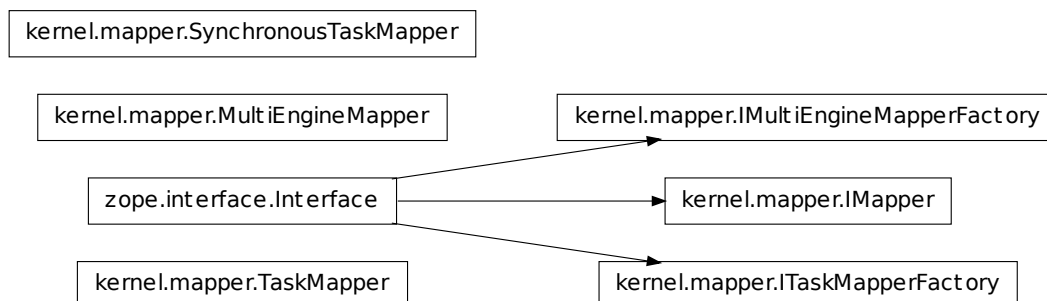
getPartition (*seq, p, q*)

joinPartitions (*listOfPartitions*)

8.64 kernel.mapper

8.64.1 Module: `kernel.mapper`

Inheritance diagram for `IPython.kernel.mapper`:



A parallelized version of Python's builtin map.

8.64.2 Classes

IMapper

class IPython.kernel.mapper.**IMapper** (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

Bases: `zope.interface.Interface`

The basic interface for a Mapper.

This defines a generic interface for mapping. The idea of this is similar to that of Python's builtin *map* function, which applies a function elementwise to a sequence.

classmethod **__init__** (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod **changed** (*originally_changed*)

We, or something we depend on, have changed

classmethod **deferred** ()

Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
```

```
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy()`
 Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor(name, default=None)`

classmethod `queryTaggedValue(tag, default=None)`
 Returns the value associated with 'tag'.

classmethod `setTaggedValue(tag, value)`
 Associates 'value' with 'key'.

classmethod `subscribe(dependent)`

classmethod `unsubscribe(dependent)`

classmethod `validateInvariants(obj, errors=None)`
 validate object to defined invariants.

classmethod `weakref(callback=None)`

IMultiEngineMapperFactory

```
class IPython.kernel.mapper.IMultiEngineMapperFactory(name, bases=(),
                                                       attrs=None,
                                                       __doc__=None,
                                                       __module__=None)
```

Bases: `zope.interface.Interface`

An interface for something that creates *IMapper* instances.

classmethod `__init__(name, bases=(), attrs=None, __doc__=None, __module__=None)`

classmethod `changed(originally_changed)`
 We, or something we depend on, have changed

classmethod `deferred()`
 Return a deferred class corresponding to the interface.

classmethod `direct(name)`

classmethod `extends(interface, strict=True)`
 Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
```

```
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
```

```

>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod `setTaggedValue` (*tag, value*)

Associates 'value' with 'key'.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

ITaskMapperFactory

```

class IPython.kernel.mapper.ITaskMapperFactory(name, bases=(), attrs=None,
__doc__=None, __module__=None)

```

Bases: `zope.interface.Interface`

An interface for something that creates *IMapper* instances.

classmethod `__init__` (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends (interface, strict=True)`

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get (name, default=None)`

Query for an attribute description

classmethod `getBases ()`

classmethod `getDescriptionFor (name)`

Return the attribute description for the given name.

classmethod `getDoc ()`

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

MultiEngineMapper

```
class IPython.kernel.mapper.MultiEngineMapper (multiengine,      dist='b',      tar-
                                              gets='all', block=True)
```

Bases: object

A Mapper for *IMultiEngine* implementers.

```
__init__ (multiengine, dist='b', targets='all', block=True)
    Create a Mapper for a multiengine.
```

The value of all arguments are used for all calls to *map*. This class allows these arguments to be set for a series of map calls.

Parameters

multiengine [*IMultiEngine* implementer] The multiengine to use for running the map commands

dist [str] The type of decomposition to use. Only block ('b') is supported currently

targets [(str, int, tuple of ints)] The engines to use in the map

block [boolean] Whether to block when the map is applied

```
map (func, *sequences)
    Apply func to *sequences elementwise. Like Python's builtin map.
```

This version is not load balanced.

SynchronousTaskMapper

```
class IPython.kernel.mapper.SynchronousTaskMapper (task_controller,
                                                    clear_before=False,
                                                    clear_after=False, retries=0,
                                                    recovery_task=None, de-
                                                    pend=None, block=True)
```

Bases: object

Make an *IBlockingTaskClient* look like an *IMapper*.

This class provides a load balanced version of *map*.

```
__init__ (task_controller, clear_before=False, clear_after=False, retries=0, recov-
          ery_task=None, depend=None, block=True)
    Create a IMapper given a IBlockingTaskClient and arguments.
```

The additional arguments are those that are common to all types of tasks and are described in the documentation for *IPython.kernel.task.BaseTask*.

Parameters

task_controller [an *IBlockingTaskClient* implementer] The *TaskController* to use for calls to *map*

map (*func*, **sequences*)

Apply *func* to **sequences* elementwise. Like Python's builtin *map*.

This version is load balanced.

TaskMapper

```
class IPython.kernel.mapper.TaskMapper (task_controller,          clear_before=False,
                                       clear_after=False,        retries=0,    recov-
                                       ery_task=None, depend=None, block=True)
```

Bases: object

Make an *ITaskController* look like an *IMapper*.

This class provides a load balanced version of *map*.

```
__init__ (task_controller, clear_before=False, clear_after=False, retries=0, recov-
         ery_task=None, depend=None, block=True)
Create a IMapper given a TaskController and arguments.
```

The additional arguments are those that are common to all types of tasks and are described in the documentation for *IPython.kernel.task.BaseTask*.

Parameters

task_controller [an *IBlockingTaskClient* implementer] The *TaskController* to use for calls to *map*

map (*func*, **sequences*)

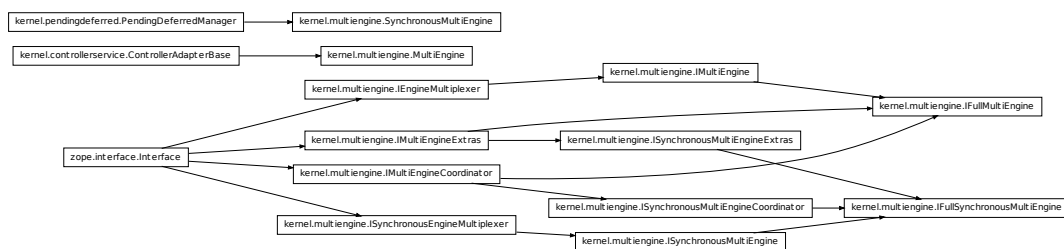
Apply *func* to **sequences* elementwise. Like Python's builtin *map*.

This version is load balanced.

8.65 kernel.multiengine

8.65.1 Module: kernel.multiengine

Inheritance diagram for `IPython.kernel.multiengine`:



Adapt the IPython ControllerServer to IMultiEngine.

This module provides classes that adapt a ControllerService to the IMultiEngine interface. This interface is a basic interactive interface for working with a set of engines where it is desired to have explicit access to each registered engine.

The classes here are exposed to the network in files like:

- multienginevanilla.py
- multienginepb.py

8.65.2 Classes

`IEngineMultiplexer`

```
class IPython.kernel.multiengine.IEngineMultiplexer (name,          bases=(),
                                                    attrs=None,
                                                    __doc__=None, __mod-
                                                    ule__=None)
```

Bases: `zope.interface.Interface`

Interface to multiple engines implementing `IEngineCore/Serialized/Queued`.

This class simply acts as a multiplexer of methods that are in the various `IEngines*` interfaces. Thus the methods here are just like those in the `IEngine*` interfaces, but with an extra first argument, `targets`. The `targets` argument can have the following forms:

- `targets = 10` # Engines are indexed by ints
- `targets = [0,1,2,3]` # A list of ints
- `targets = 'all'` # A string to indicate all targets

If `targets` is bad in any way, an `InvalidEngineID` will be raised. This includes engines not being registered.

All `IEngineMultiplexer` multiplexer methods must return a `Deferred` to a list with length equal to the number of targets. The elements of the list will correspond to the return of the corresponding `IEngine` method.

Failures are aggressive, meaning that if an action fails for any target, the overall action will fail immediately with that Failure.

Parameters

targets [int, list of ints, or 'all'] Engine ids the action will apply to.

Returns `Deferred` to a list of results for each engine.

Exception

InvalidEngineID If the `targets` argument is bad or engines aren't registered.

NoEnginesRegistered If there are no engines registered and `targets='all'`

```
classmethod __init__ (name, bases=(), attrs=None, __doc__=None, __module__=None)
```

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with ‘tag’.

classmethod `setTaggedValue (tag, value)`

Associates ‘value’ with ‘key’.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod weakref (*callback=None*)

IFullMultiEngine

class IPython.kernel.multiengine.**IFullMultiEngine** (*name, bases=(), attrs=None, __doc__=None, __module__=None, __weakref__=None*)

Bases: IPython.kernel.multiengine.IMultiEngine, IPython.kernel.multiengine.IMultiEngineCoordinator, IPython.kernel.multiengine.IMultiEngineExtras

classmethod __init__ (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod changed (*originally_changed*)

We, or something we depend on, have changed

classmethod deferred ()

Return a deferred class corresponding to the interface.

classmethod direct (*name*)

classmethod extends (*interface, strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
```

```
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static providedBy ()
 Test whether an interface is implemented by the specification

classmethod queryDescriptionFor (*name*, *default=None*)

classmethod queryTaggedValue (*tag*, *default=None*)
 Returns the value associated with 'tag'.

classmethod setTaggedValue (*tag*, *value*)
 Associates 'value' with 'key'.

classmethod subscribe (*dependent*)

classmethod unsubscribe (*dependent*)

classmethod validateInvariants (*obj*, *errors=None*)
 validate object to defined invariants.

classmethod weakref (*callback=None*)

IFullSynchronousMultiEngine

```
class IPython.kernel.multiengine.IFullSynchronousMultiEngine (name,
                                                                bases=(),
                                                                attrs=None,
                                                                __doc__=None,
                                                                __mod-
                                                                ule__=None)
Bases: IPython.kernel.multiengine.ISynchronousMultiEngine,
        IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator,
        IPython.kernel.multiengine.ISynchronousMultiEngineExtras
classmethod __init__ (name, bases=(), attrs=None, __doc__=None, __module__=None)
classmethod changed (originally_changed)
    We, or something we depend on, have changed
classmethod deferred ()
    Return a deferred class corresponding to the interface.
classmethod direct (name)
classmethod extends (interface, strict=True)
    Does the specification extend the given interface?
    Test whether an interface in the specification extends the given interface
```

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...

```

```
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod isEqualOrExtendedBy (*other*)

Same interface or extends?

static isOrExtends ()

Test whether a specification is or extends another

classmethod names (*all=False*)

Return the attribute names defined by the interface.

classmethod namesAndDescriptions (*all=False*)

Return attribute names and descriptions defined by interface.

static providedBy ()

Test whether an interface is implemented by the specification

classmethod queryDescriptionFor (*name, default=None*)

classmethod queryTaggedValue (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod setTaggedValue (*tag, value*)

Associates 'value' with 'key'.

classmethod subscribe (*dependent*)

classmethod unsubscribe (*dependent*)

classmethod validateInvariants (*obj, errors=None*)

validate object to defined invariants.

classmethod weakref (*callback=None*)

IMultiEngine

```

class IPython.kernel.multiengine.IMultiEngine(name, bases=(), attrs=None,
                                              __doc__=None, __module__=None,
                                              __weakref__=None)

```

Bases: IPython.kernel.multiengine.IEngineMultiplexer

A controller that exposes an explicit interface to all of its engines.

This is the primary interface for interactive usage.

classmethod __init__ (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with ‘tag’.

classmethod `setTaggedValue (tag, value)`

Associates ‘value’ with ‘key’.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod weakref (*callback=None*)

IMultiEngineCoordinator

```
class IPython.kernel.multiengine.IMultiEngineCoordinator(name, bases=(),
                                                         attrs=None,
                                                         __doc__=None,
                                                         __module__=None,
                                                         __weakref__=None)
```

Bases: `zope.interface.Interface`

Methods that work on multiple engines explicitly.

classmethod __init__ (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod changed (*originally_changed*)

We, or something we depend on, have changed

classmethod deferred ()

Return a deferred class corresponding to the interface.

classmethod direct (*name*)

classmethod extends (*interface, strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
```

```

>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod **namesAndDescriptions** (*all=False*)
Return attribute names and descriptions defined by interface.

static **providedBy** ()
Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name, default=None*)

classmethod **queryTaggedValue** (*tag, default=None*)
Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)
Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj, errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

IMultiEngineExtras

```
class IPython.kernel.multiengine.IMultiEngineExtras (name, bases=(),  
                                                    attrs=None,  
                                                    __doc__=None, __module__=None)
```

Bases: `zope.interface.Interface`

classmethod **__init__** (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod **changed** (*originally_changed*)
We, or something we depend on, have changed

classmethod **deferred** ()
Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface, strict=True*)
Does the specification extend the given interface?
Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
```



```

>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...

```

```
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod `setTaggedValue` (*tag, value*)

Associates 'value' with 'key'.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

ISynchronousEngineMultiplexer

```
class IPython.kernel.multiengine.ISynchronousEngineMultiplexer (name,
                                                                bases=(),
                                                                at-
                                                                trs=None,
                                                                __doc__=None,
                                                                __mod-
                                                                ule__=None)
```

Bases: `zope.interface.Interface`

classmethod `__init__` (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod deferred()

Return a deferred class corresponding to the interface.

classmethod direct (*name*)

classmethod extends (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod get (*name*, *default=None*)

Query for an attribute description

classmethod getBases ()

classmethod getDescriptionFor (*name*)

Return the attribute description for the given name.

classmethod getDoc ()

Returns the documentation for the object.

classmethod getName ()

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

ISynchronousMultiEngine

```
class IPython.kernel.multiengine.ISynchronousMultiEngine(name, bases=(),
                                                         attrs=None,
                                                         __doc__=None,
                                                         __module__=None,
                                                         __weakref__=None)
```

Bases: `IPython.kernel.multiengine.ISynchronousEngine``Multiplexer`

Synchronous, two-phase version of `IMultiEngine`.

Methods in this interface are identical to those of `IMultiEngine`, but they take one additional argument:

`execute(lines, targets='all')` -> `execute(lines, targets='all', block=True)`

Parameters

block [boolean] Should the method return a deferred to a deferredID or the actual result. If `block=False` a deferred to a deferredID is returned and the user must call `get_pending_deferred` at a later point. If `block=True`, a deferred to the actual result comes back.

classmethod `__init__` (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
```

```
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static isOrExtends ()
 Test whether a specification is or extends another

classmethod names (*all=False*)
 Return the attribute names defined by the interface.

classmethod namesAndDescriptions (*all=False*)
 Return attribute names and descriptions defined by interface.

static providedBy ()
 Test whether an interface is implemented by the specification

classmethod queryDescriptionFor (*name, default=None*)

classmethod queryTaggedValue (*tag, default=None*)
 Returns the value associated with 'tag'.

classmethod setTaggedValue (*tag, value*)
 Associates 'value' with 'key'.

classmethod subscribe (*dependent*)

classmethod unsubscribe (*dependent*)

classmethod validateInvariants (*obj, errors=None*)
 validate object to defined invariants.

classmethod weakref (*callback=None*)

ISynchronousMultiEngineCoordinator

```
class IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator(name,
                                                                    bases=(),
                                                                    at-
                                                                    trs=None,
                                                                    __doc__=None,
                                                                    __mod-
                                                                    ule__=None)
```

Bases: `IPython.kernel.multiengine.IMultiEngineCoordinator`

Methods that work on multiple engines explicitly.

classmethod __init__ (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod changed (*originally_changed*)
 We, or something we depend on, have changed

classmethod deferred ()
 Return a deferred class corresponding to the interface.

classmethod direct (*name*)

classmethod extends (*interface, strict=True*)
 Does the specification extend the given interface?
 Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument

is neither a class nor a callable.

classmethod interfaces ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod isEqualOrExtendedBy (other)

Same interface or extends?

static isOrExtends ()

Test whether a specification is or extends another

classmethod names (all=False)

Return the attribute names defined by the interface.

classmethod namesAndDescriptions (all=False)

Return attribute names and descriptions defined by interface.

static providedBy ()

Test whether an interface is implemented by the specification

classmethod queryDescriptionFor (name, default=None)

classmethod queryTaggedValue (tag, default=None)

Returns the value associated with 'tag'.

classmethod setTaggedValue (tag, value)

Associates 'value' with 'key'.

classmethod subscribe (dependent)

classmethod unsubscribe (dependent)

classmethod validateInvariants (obj, errors=None)

validate object to defined invariants.

classmethod weakref (callback=None)

ISynchronousMultiEngineExtras

```
class IPython.kernel.multiengine.ISynchronousMultiEngineExtras (name,
                                                                bases=(),
                                                                at-
                                                                trs=None,
                                                                __doc__=None,
                                                                __mod-
                                                                ule__=None)
```

Bases: `IPython.kernel.multiengine.IMultiEngineExtras`

classmethod `__init__` (*name*, *bases*=(), *attrs*=None, *__doc__*=None, *__module__*=None)

classmethod `changed` (*originally_changed*)
We, or something we depend on, have changed

classmethod `deferred` ()
Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict*=True)
Does the specification extend the given interface?
Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
```

```

1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static **providedBy** ()
Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name*, *default=None*)

classmethod **queryTaggedValue** (*tag*, *default=None*)
Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag*, *value*)
Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj*, *errors=None*)
validate object to defined invariants.

classmethod **weakref** (*callback=None*)

MultiEngine

class IPython.kernel.multiengine.**MultiEngine** (*controller*)
Bases: IPython.kernel.controllerservice.ControllerAdapterBase

The representation of a ControllerService as a IMultiEngine.

Although it is not implemented currently, this class would be where a client/notification API is implemented. It could inherit from something like results.NotifierParent and then use the notify method to send notifications.

__init__ (*controller*)

clear_properties (*targets='all'*)

clear_queue (*targets='all'*)

del_properties (*keys*, *targets='all'*)

engineList (*targets*)
Parse the targets argument into a list of valid engine objects.

Parameters

targets [int, list of ints or 'all'] The targets argument to be parsed.

Returns List of engine objects.

Exception

InvalidEngineID If targets is not valid or if an engine is not registered.

execute (*lines*, *targets='all'*)

get_ids ()

get_properties (*keys=None*, *targets='all'*)

```

get_result (i=None, targets='all')
has_properties (keys, targets='all')
keys (targets='all')
kill (controller=False, targets='all')
on_n_engines_registered_do (n,f, *args, **kwargs)
on_register_engine_do (f, includeID, *args, **kwargs)
on_register_engine_do_not (f)
on_unregister_engine_do (f, includeID, *args, **kwargs)
on_unregister_engine_do_not (f)
pull (keys, targets='all')
pull_function (keys, targets='all')
pull_serialized (keys, targets='all')
push (ns, targets='all')
push_function (ns, targets='all')
push_serialized (namespace, targets='all')
queue_status (targets='all')
register_engine (remoteEngine, id=None, ip=None, port=None, pid=None)
reset (targets='all')
set_properties (properties, targets='all')
unregister_engine (id)

```

SynchronousMultiEngine

```

class IPython.kernel.multiengine.SynchronousMultiEngine (multiengine)
    Bases: IPython.kernel.pendingdeferred.PendingDeferredManager

```

Adapt an *IMultiEngine* -> *ISynchronousMultiEngine*

Warning, this class uses a decorator that currently uses ****kwargs**. Because of this block must be passed as a kwarg, not positionally.

```

__init__ (multiengine)

clear_pending_deferreds ()
    Remove all the deferreds I am tracking.

clear_properties (pdm, *args, **kwargs)

clear_queue (pdm, *args, **kwargs)

del_properties (pdm, *args, **kwargs)

```

delete_pending_deferred(*deferred_id*)

Remove a deferred I am tracking and add a null Errback.

Parameters

deferredID [str] The id of a deferred that I am tracking.

execute(*pdm, *args, **kwargs*)

get_deferred_id()

get_ids()

Return a list of registered engine ids.

Never use the two phase block/non-block stuff for this.

get_pending_deferred(*deferred_id, block*)

get_properties(*pdm, *args, **kwargs*)

get_result(*pdm, *args, **kwargs*)

has_properties(*pdm, *args, **kwargs*)

keys(*pdm, *args, **kwargs*)

kill(*pdm, *args, **kwargs*)

pull(*pdm, *args, **kwargs*)

pull_function(*pdm, *args, **kwargs*)

pull_serialized(*pdm, *args, **kwargs*)

push(*pdm, *args, **kwargs*)

push_function(*pdm, *args, **kwargs*)

push_serialized(*pdm, *args, **kwargs*)

queue_status(*pdm, *args, **kwargs*)

quick_has_id(*deferred_id*)

reset(*pdm, *args, **kwargs*)

save_pending_deferred(*d, deferred_id=None*)

Save the result of a deferred for later retrieval.

This works even if the deferred has not fired.

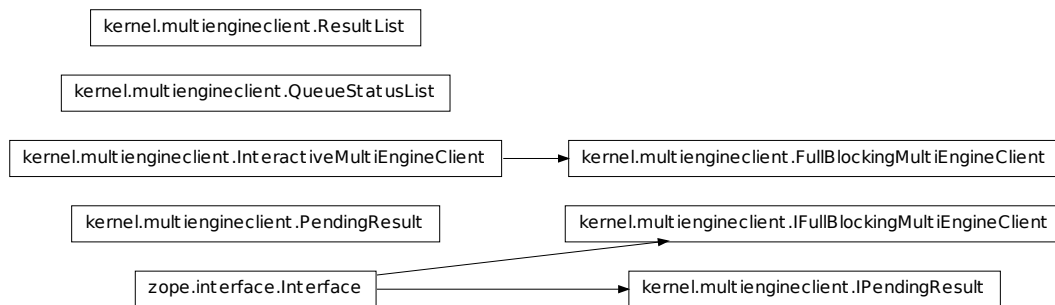
Only callbacks and errbacks applied to *d* before this method is called will be called no the final result.

set_properties(*pdm, *args, **kwargs*)

8.66 kernel.multiengineclient

8.66.1 Module: kernel.multiengineclient

Inheritance diagram for IPython.kernel.multiengineclient:



General Classes for IMultiEngine clients.

8.66.2 Classes

FullBlockingMultiEngineClient

class IPython.kernel.multiengineclient.**FullBlockingMultiEngineClient** (*smultiengine*)
 Bases: IPython.kernel.multiengineclient.InteractiveMultiEngineClient

A blocking client to the *IMultiEngine* controller interface.

This class allows users to use a set of engines for a parallel computation through the *IMultiEngine* interface. In this interface, each engine has a specific id (an int) that is used to refer to the engine, run code on it, etc.

__init__ (*smultiengine*)

activate ()

Make this *MultiEngineClient* active for parallel magic commands.

IPython has a magic command syntax to work with *MultiEngineClient* objects. In a given IPython session there is a single active one. While there can be many *MultiEngineClient* created and used by the user, there is only one active one. The active *MultiEngineClient* is used whenever the magic commands %px and %autopx are used.

The activate() method is called on a given *MultiEngineClient* to make it active. Once this has been done, the magic commands can be used.

barrier (*pendingResults*)

Synchronize a set of *PendingResults*.

This method is a synchronization primitive that waits for a set of *PendingResult* objects to complete. More specifically, *barrier* does the following.

- The ‘*PendingResult*’s are sorted by *result_id*.
- The *get_result* method is called for each *PendingResult* sequentially with *block=True*.
- If a *PendingResult* gets a result that is an exception, it is trapped and can be re-raised later by calling *get_result* again.
- The ‘*PendingResult*’s are flushed from the controller.

After *barrier* has been called on a *PendingResult*, its results can be retrieved by calling *get_result* again or accessing the *r* attribute of the instance.

benchmark (*push_size=10000*)

Run performance benchmarks for the current IPython cluster.

This method tests both the latency of sending command and data to the engines as well as the throughput of sending large objects to the engines using *push*. The latency is measured by having one or more engines execute the command ‘*pass*’. The throughput is measure by sending an NumPy array of size *push_size* to one or more engines.

These benchmarks will vary widely on different hardware and networks and thus can be used to get an idea of the performance characteristics of a particular configuration of an IPython controller and engines.

This function is not testable within our current testing framework.

clear_pending_results ()

Clear all pending deferreds/results from the controller.

For each *PendingResult* that is created by this client, the controller holds on to the result for that *PendingResult*. This can be a problem if there are a large number of *PendingResult* objects that are created.

Once the result of the *PendingResult* has been retrieved, the result is removed from the controller, but if a user doesn’t get a result (they just ignore the *PendingResult*) the result is kept forever on the controller. This method allows the user to clear out all un-retrieved results on the controller.

clear_properties (*targets=None, block=None*)

clear_queue (*targets=None, block=None*)

Clear out the controller’s queue for an engine.

The controller maintains a queue for each engine. This clear it out.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

del_properties (*keys, targets=None, block=None*)

execute (*lines, targets=None, block=None*)

Execute code on a set of engines.

Parameters

lines [str] The Python code to execute as a string

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

flush()

Clear all pending deferreds/results from the controller.

For each *PendingResult* that is created by this client, the controller holds on to the result for that *PendingResult*. This can be a problem if there are a large number of *PendingResult* objects that are created.

Once the result of the *PendingResult* has been retrieved, the result is removed from the controller, but if a user doesn't get a result (they just ignore the *PendingResult*) the result is kept forever on the controller. This method allows the user to clear out all un-retrieved results on the controller.

gather (*key*, *dist*='b', *targets*=None, *block*=None)

Gather a partitioned sequence on a set of engines as a single local seq.

get_ids()

Returns the ids of currently registered engines.

get_pending_deferred (*deferredID*, *block*)

get_properties (*keys*=None, *targets*=None, *block*=None)

get_result (*i*=None, *targets*=None, *block*=None)

Get a previous result.

When code is executed in an engine, a dict is created and returned. This method retrieves that dict for previous commands.

Parameters

i [int] The number of the result to get

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

has_properties (*keys*, *targets*=None, *block*=None)

keys (*targets*=None, *block*=None)

Get a list of all the variables in an engine's namespace.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

kill (*controller=False, targets=None, block=None*)

Kill the engines and controller.

This method is used to stop the engine and controller by calling *reactor.stop*.

Parameters

controller [boolean] If True, kill the engines and controller. If False, just the engines

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

map (*func, *sequences*)

A parallel version of Python's builtin *map* function.

This method applies a function to sequences of arguments. It follows the same syntax as the builtin *map*.

This method creates a mapper objects by calling *self.mapper* with no arguments and then uses that mapper to do the mapping. See the documentation of *mapper* for more details.

mapper (*dist='b', targets='all', block=None*)

Create a mapper object that has a *map* method.

This method returns an object that implements the *IMapper* interface. This method is a factory that is used to control how the map happens.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

parallel (*dist='b', targets=None, block=None*)

A decorator that turns a function into a parallel function.

This can be used as:

```
@parallel() def f(x, y)
```

```
...
```

```
f(range(10), range(10))
```

This causes *f*(0,0), *f*(1,1), ... to be called in parallel.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

pull (*keys*, *targets=None*, *block=None*)

Pull Python objects by key out of engines namespaces.

Parameters

keys [str or list of str] The names of the variables to be pulled

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

pull_function (*keys*, *targets=None*, *block=None*)

Pull a Python function from an engine.

This method is used to pull a Python function from an engine. Closures are not supported.

Parameters

keys [str or list of str] The names of the functions to be pulled

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

pull_serialized (*keys*, *targets=None*, *block=None*)

push (*namespace*, *targets=None*, *block=None*)

Push a dictionary of keys and values to engines namespace.

Each engine has a persistent namespace. This method is used to push Python objects into that namespace.

The objects in the namespace must be pickleable.

Parameters

namespace [dict] A dict that contains Python objects to be injected into the engine persistent namespace.

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

push_function (*namespace*, *targets=None*, *block=None*)

Push a Python function to an engine.

This method is used to push a Python function to an engine. This method can then be used in code on the engines. Closures are not supported.

Parameters

namespace [dict] A dict whose values are the functions to be pushed. The keys give that names that the function will appear as in the engines namespace.

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

push_serialized (*namespace*, *targets=None*, *block=None*)

queue_status (*targets=None*, *block=None*)

Get the status of an engines queue.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

raw_map (*func*, *seq*, *dist='b'*, *targets=None*, *block=None*)

A parallelized version of Python's builtin map.

This has a slightly different syntax than the builtin *map*. This is needed because we need to have keyword arguments and thus can't use **args* to capture all the sequences. Instead, they must be passed in a list or tuple.

`raw_map(func, seqs) -> map(func, seqs[0], seqs[1], ...)`

Most users will want to use parallel functions or the *mapper* and *map* methods for an API that follows that of the builtin *map*.

reset (*targets=None*, *block=None*)

Reset an engine.

This method clears out the namespace of an engine.

Parameters

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

run (*filename*, *targets=None*, *block=None*)

Run a Python code in a file on the engines.

Parameters

filename [str] The name of the local file to run

targets [id or list of ids] The engine to use for the execution

block [boolean] If False, this method will return the actual result. If False, a *PendingResult* is returned which can be used to get the result at a later time.

scatter (*key*, *seq*, *dist='b'*, *flatten=False*, *targets=None*, *block=None*)

Partition a Python sequence and send the partitions to a set of engines.

set_properties (*properties*, *targets=None*, *block=None*)

zip_pull (*keys*, *targets=None*, *block=None*)

IFullBlockingMultiEngineClient

```
class IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient (name,
                                                                    bases=(),
                                                                    at-
                                                                    trs=None,
                                                                    __doc__=None,
                                                                    __mod-
                                                                    ule__=None)
```

Bases: `zope.interface.Interface`

classmethod `__init__` (*name*, *bases*=(), *attrs*=None, *__doc__*=None, *__module__*=None)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict*=True)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
```

```
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static **providedBy** ()
 Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name*, *default=None*)

classmethod **queryTaggedValue** (*tag*, *default=None*)
 Returns the value associated with ‘tag’.

classmethod **setTaggedValue** (*tag*, *value*)
 Associates ‘value’ with ‘key’.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj*, *errors=None*)
 validate object to defined invariants.

classmethod **weakref** (*callback=None*)

IPendingResult

```
class IPython.kernel.multiengineclient.IPendingResult (name, bases=(),
                                                    attrs=None,
                                                    __doc__=None,
                                                    __module__=None)
```

Bases: `zope.interface.Interface`

A representation of a result that is pending.

This class is similar to Twisted’s *Deferred* object, but is designed to be used in a synchronous context.

classmethod **__init__** (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod **changed** (*originally_changed*)
 We, or something we depend on, have changed

classmethod **deferred** ()
 Return a deferred class corresponding to the interface.

classmethod **direct** (*name*)

classmethod **extends** (*interface*, *strict=True*)
 Does the specification extend the given interface?
 Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
```

```
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
```



```

...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with ‘tag’.

classmethod `setTaggedValue` (*tag, value*)

Associates ‘value’ with ‘key’.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

InteractiveMultiEngineClient

class IPython.kernel.multiengineclient.**InteractiveMultiEngineClient**

Bases: object

A mixin class that add a few methods to a multiengine client.

The methods in this mixin class are designed for interactive usage.

`__init__` ()

x.`__init__`(...) initializes x; see x.`__class__`.`__doc__` for signature

`activate` ()

Make this *MultiEngineClient* active for parallel magic commands.

IPython has a magic command syntax to work with *MultiEngineClient* objects. In a given IPython session there is a single active one. While there can be many *MultiEngineClient* created and used by the user, there is only one active one. The active *MultiEngineClient* is used whenever the magic commands `%px` and `%autopx` are used.

The `activate()` method is called on a given *MultiEngineClient* to make it active. Once this has been done, the magic commands can be used.

PendingResult

class IPython.kernel.multiengineclient.**PendingResult** (*client, result_id*)

Bases: object

A representation of a result that is not yet ready.

A user should not create a *PendingResult* instance by hand.

Methods:

- `get_result`
- `add_callback`

Properties:

- `r`

__init__ (*client, result_id*)

Create a *PendingResult* with a `result_id` and a client instance.

The client should implement `_getPendingResult(result_id, block)`.

add_callback (*f, *args, **kwargs*)

Add a callback that is called with the result.

If the original result is `result`, adding a callback will cause `f(result, *args, **kwargs)` to be returned instead. If multiple callbacks are registered, they are chained together: the result of one is passed to the next and so on.

Unlike Twisted's *Deferred* object, there is no errback chain. Thus any exception raised will not be caught and handled. User must catch these by hand when calling `get_result`.

get_result (*default=None, block=True*)

Get a result that is pending.

This method will connect to an *IMultiEngine* adapted controller and see if the result is ready. If the action triggers an exception raise it and record it. This method records the result/exception once it is retrieved. Calling `get_result` again will get this cached result or will re-raise the exception. The `.r` attribute is a property that calls `get_result` with `block=True`.

Parameters

default The value to return if the result is not ready.

block [boolean] Should I block for the result.

Returns The actual result or the default value.

r

This property is a shortcut to a `get_result(block=True)`.

QueueStatusList

class IPython.kernel.multiengineclient.**QueueStatusList**

Bases: list

A subclass of list that pretty prints the output of `queue_status`.

__init__()

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature

append

`L.append(object)` – append object to end

count

`L.count(value)` -> integer – return number of occurrences of value

extend

`L.extend(iterable)` – extend list by appending elements from the iterable

index

`L.index(value, [start, [stop]])` -> integer – return first index of value. Raises `ValueError` if the value is not present.

insert

`L.insert(index, object)` – insert object before index

pop

`L.pop([index])` -> item – remove and return item at index (default last). Raises `IndexError` if list is empty or index is out of range.

remove

`L.remove(value)` – remove first occurrence of value. Raises `ValueError` if the value is not present.

reverse

`L.reverse()` – reverse *IN PLACE*

sort

`L.sort(cmp=None, key=None, reverse=False)` – stable sort *IN PLACE*; `cmp(x, y)` -> -1, 0, 1

ResultList

class IPython.kernel.multiengineclient.**ResultList**

Bases: list

A subclass of list that pretty prints the output of `execute/get_result`.

__init__()

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature

append

L.append(object) – append object to end

count

L.count(value) -> integer – return number of occurrences of value

extend

L.extend(iterable) – extend list by appending elements from the iterable

index

L.index(value, [start, [stop]]) -> integer – return first index of value. Raises ValueError if the value is not present.

insert

L.insert(index, object) – insert object before index

pop

L.pop([index]) -> item – remove and return item at index (default last). Raises IndexError if list is empty or index is out of range.

remove

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse

L.reverse() – reverse *IN PLACE*

sort

L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

8.66.3 Function

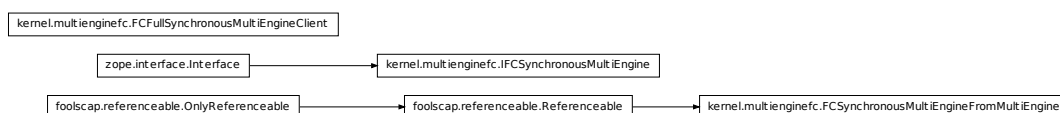
`IPython.kernel.multiengineclient.wrapResultList(result)`

A function that wraps the output of `execute/get_result` -> `ResultList`.

8.67 kernel.multienginefc

8.67.1 Module: kernel.multienginefc

Inheritance diagram for `IPython.kernel.multienginefc`:



Expose the multiengine controller over the Foolscape network protocol.

8.67.2 Classes

FCFullSynchronousMultiEngineClient

class IPython.kernel.multienginefc.**FCFullSynchronousMultiEngineClient** (*remote_reference*)

Bases: object

__init__ (*remote_reference*)

adapt_to_blocking_client ()

clear_pending_deferreds ()

clear_properties (*targets='all', block=True*)

clear_queue (*targets='all', block=True*)

del_properties (*keys, targets='all', block=True*)

execute (*lines, targets='all', block=True*)

gather (*key, dist='b', targets='all', block=True*)

get_ids ()

get_pending_deferred (*deferredID, block=True*)

get_properties (*keys=None, targets='all', block=True*)

get_result (*i=None, targets='all', block=True*)

has_properties (*keys, targets='all', block=True*)

keys (*targets='all', block=True*)

kill (*controller=False, targets='all', block=True*)

map (*func, *sequences*)

A parallel version of Python's builtin *map* function.

This method applies a function to sequences of arguments. It follows the same syntax as the builtin *map*.

This method creates a mapper objects by calling *self.mapper* with no arguments and then uses that mapper to do the mapping. See the documentation of *mapper* for more details.

mapper (*dist='b', targets='all', block=True*)

Create a mapper object that has a *map* method.

This method returns an object that implements the *IMapper* interface. This method is a factory that is used to control how the map happens.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

parallel (*dist='b', targets='all', block=True*)

A decorator that turns a function into a parallel function.

This can be used as:

```
@parallel() def f(x, y)
```

...

```
f(range(10), range(10))
```

This causes `f(0,0)`, `f(1,1)`, ... to be called in parallel.

Parameters

dist [str] What decomposition to use, 'b' is the only one supported currently

targets [str, int, sequence of ints] Which engines to use for the map

block [boolean] Should calls to *map* block or not

pull (*keys, targets='all', block=True*)

pull_function (*keys, targets='all', block=True*)

pull_serialized (*keys, targets='all', block=True*)

push (*namespace, targets='all', block=True*)

push_function (*namespace, targets='all', block=True*)

push_serialized (*namespace, targets='all', block=True*)

queue_status (*targets='all', block=True*)

raw_map (*func, sequences, dist='b', targets='all', block=True*)

A parallelized version of Python's builtin map.

This has a slightly different syntax than the builtin *map*. This is needed because we need to have keyword arguments and thus can't use **args* to capture all the sequences. Instead, they must be passed in a list or tuple.

```
raw_map(func, seqs) -> map(func, seqs[0], seqs[1], ...)
```

Most users will want to use parallel functions or the *mapper* and *map* methods for an API that follows that of the builtin *map*.

reset (*targets='all', block=True*)

run (*fname, targets='all', block=True*)

scatter (*key, seq, dist='b', flatten=False, targets='all', block=True*)

set_properties (*properties, targets='all', block=True*)

unpackage (*r*)

zip_pull (*keys, targets='all', block=True*)

FCSynchronousMultiEngineFromMultiEngine

class IPython.kernel.multienginefc.**FCSynchronousMultiEngineFromMultiEngine** (*multiengine*)

Bases: `foolscap.referenceable.Referenceable`

Adapt *IMultiEngine* -> *ISynchronousMultiEngine* -> *IFCSynchronousMultiEngine*.

__init__ (*multiengine*)

doRemoteCall (*methodname*, *args*, *kwargs*)

getInterface ()

getInterfaceName ()

packageFailure (*f*)

packageSuccess (*obj*)

processUniqueID ()

remote_clear_pending_deferreds (**args*, ***kwargs*)

remote_clear_properties (**args*, ***kwargs*)

remote_clear_queue (**args*, ***kwargs*)

remote_del_properties (**args*, ***kwargs*)

remote_execute (**args*, ***kwargs*)

remote_get_client_name ()

remote_get_ids ()

Get the ids of the registered engines.

This method always blocks.

remote_get_pending_deferred (**args*, ***kwargs*)

remote_get_properties (**args*, ***kwargs*)

remote_get_result (**args*, ***kwargs*)

remote_has_properties (**args*, ***kwargs*)

remote_keys (**args*, ***kwargs*)

remote_kill (**args*, ***kwargs*)

remote_pull (**args*, ***kwargs*)

remote_pull_function (**args*, ***kwargs*)

remote_pull_serialized (**args*, ***kwargs*)

remote_push (**args*, ***kwargs*)

remote_push_function (**args*, ***kwargs*)

remote_push_serialized (**args*, ***kwargs*)

```
remote_queue_status (*args, **kwargs)
remote_reset (*args, **kwargs)
remote_set_properties (*args, **kwargs)
```

IFCSynchronousMultiEngine

```
class IPython.kernel.multienginefc.IFCSynchronousMultiEngine (name,
                                                                bases=(),
                                                                attrs=None,
                                                                __doc__=None,
                                                                __module__=None,
                                                                __module__=None)
```

Bases: `zope.interface.Interface`

Foolscap interface to *ISynchronousMultiEngine*.

The methods in this interface are similar to those of *ISynchronousMultiEngine*, but their arguments and return values are pickled if they are not already simple Python types that can be send over XML-RPC.

See the documentation of *ISynchronousMultiEngine* and *IMultiEngine* for documentation about the methods.

Most methods in this interface act like the *ISynchronousMultiEngine* versions and can be called in blocking or non-blocking mode.

classmethod `__init__` (*name*, *bases*=(), *attrs*=None, *__doc__*=None, *__module__*=None)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict*=True)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
```



```

>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()

```

```
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod `setTaggedValue` (*tag, value*)

Associates 'value' with 'key'.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

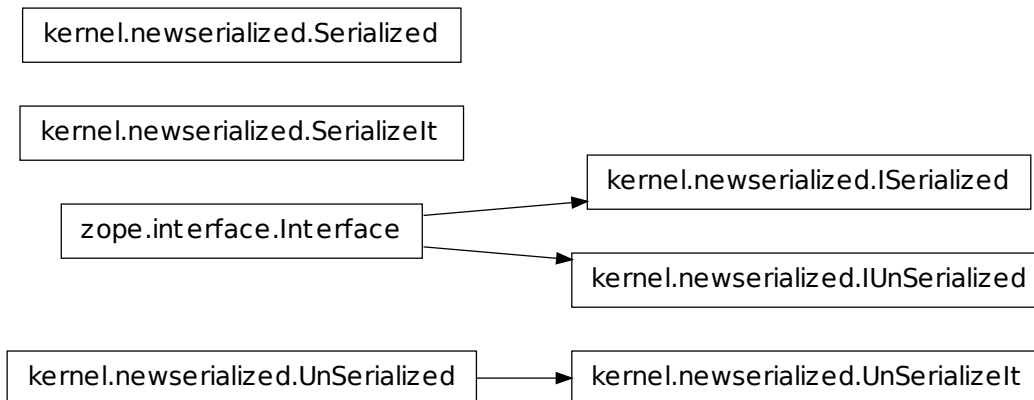
8.67.3 Function

`IPython.kernel.multienginefc.packageResult` (*wrappedMethod*)

8.68 `kernel.newserialized`

8.68.1 Module: `kernel.newserialized`

Inheritance diagram for `IPython.kernel.newserialized`:



Refactored serialization classes and interfaces.

8.68.2 Classes

ISerialized

```
class IPython.kernel.newserialized.ISerialized(name, bases=(), attrs=None,
                                              __doc__=None, __module__=None,
                                              __le__=None)
```

Bases: `zope.interface.Interface`

classmethod `__init__`(*name*, *bases*=(), *attrs*=None, *__doc__*=None, *__module__*=None)

classmethod `changed`(*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred`()

Return a deferred class corresponding to the interface.

classmethod `direct`(*name*)

classmethod `extends`(*interface*, *strict*=True)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>>
```

```
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]

```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod `setTaggedValue` (*tag, value*)

Associates 'value' with 'key'.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

IUnSerialized

```

class IPython.kernel.newserialized.IUnSerialized(name, bases=(), attrs=None,
                                                  __doc__=None, __module__=None)

```

Bases: `zope.interface.Interface`

classmethod `__init__` (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred()`

Return a deferred class corresponding to the interface.

classmethod `direct (name)`

classmethod `extends (interface, strict=True)`

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get (name, default=None)`

Query for an attribute description

classmethod `getBases()`

classmethod `getDescriptionFor (name)`

Return the attribute description for the given name.

classmethod `getDoc()`

Returns the documentation for the object.

classmethod `getName()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

SerializeIt

```
class IPython.kernel.newserialized.SerializeIt (unSerialized)
    Bases: object
    __init__ (unSerialized)
    getData ()
    getDataSize (units=1000000.0)
    getMetadata ()
    getTypeDescriptor ()
```

Serialized

```
class IPython.kernel.newserialized.Serialized (data, typeDescriptor, meta-
                                                data={})
    Bases: object
    __init__ (data, typeDescriptor, metadata={})
    getData ()
    getDataSize (units=1000000.0)
    getMetadata ()
    getTypeDescriptor ()
```

UnSerializeIt

```
class IPython.kernel.newserialized.UnSerializeIt (serialized)
    Bases: IPython.kernel.newserialized.UnSerialized
    __init__ (serialized)
    getObject ()
```

UnSerialized

```
class IPython.kernel.newserialized.UnSerialized (obj)
    Bases: object
    __init__ (obj)
    getObject ()
```


8.68.3 Functions

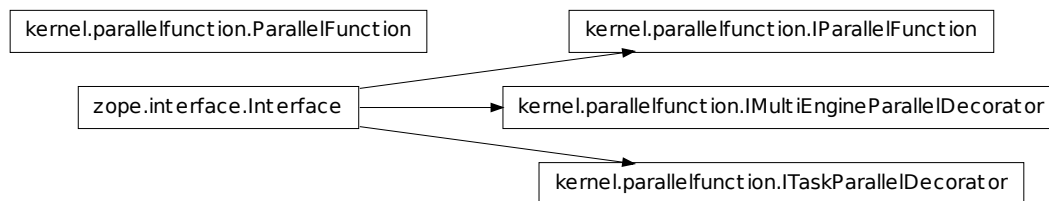
`IPython.kernel.newserialized.serialize(obj)`

`IPython.kernel.newserialized.unserialize(serialized)`

8.69 kernel.parallelfunction

8.69.1 Module: kernel.parallelfunction

Inheritance diagram for `IPython.kernel.parallelfunction`:



A parallelized function that does scatter/execute/gather.

8.69.2 Classes

IMultiEngineParallelDecorator

```

class IPython.kernel.parallelfunction.IMultiEngineParallelDecorator(name,
                                                                    bases=(),
                                                                    attrs=None,
                                                                    __doc__=None,
                                                                    __module__=None,
                                                                    __module__=None)
  
```

Bases: `zope.interface.Interface`

A decorator that creates a parallel function.

classmethod `__init__`(name, bases=(), attrs=None, __doc__=None, __module__=None)

classmethod `changed`(originally_changed)

We, or something we depend on, have changed

classmethod `deferred`()

Return a deferred class corresponding to the interface.

classmethod `direct`(name)

classmethod `extends (interface, strict=True)`

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1
```

classmethod `get (name, default=None)`

Query for an attribute description

classmethod `getBases ()`

classmethod `getDescriptionFor (name)`

Return the attribute description for the given name.

classmethod `getDoc ()`

Returns the documentation for the object.

classmethod `getName ()`

Returns the name of the object.

classmethod `getTaggedValue (tag)`

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags ()`

Returns a list of all tags.

static `implementedBy ()`

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces ()`

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy (other)`

Same interface or extends?

static `isOrExtends ()`

Test whether a specification is or extends another

classmethod `names (all=False)`

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions (all=False)`

Return attribute names and descriptions defined by interface.

static `providedBy ()`

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor (name, default=None)`

classmethod `queryTaggedValue (tag, default=None)`

Returns the value associated with 'tag'.

classmethod `setTaggedValue (tag, value)`

Associates 'value' with 'key'.

classmethod `subscribe (dependent)`

classmethod `unsubscribe (dependent)`

classmethod `validateInvariants (obj, errors=None)`

validate object to defined invariants.

classmethod `weakref (callback=None)`

`IParallelFunction`

```
class IPython.kernel.parallelfunction.IParallelFunction(name, bases=(),
                                                         attrs=None,
                                                         __doc__=None,
                                                         __module__=None,
                                                         __doc__=None)
```

Bases: `zope.interface.Interface`

classmethod `__init__` (*name*, *bases=()*, *attrs=None*, *__doc__=None*, *__module__=None*)

classmethod `changed` (*originally_changed*)
We, or something we depend on, have changed

classmethod `deferred` ()
Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface*, *strict=True*)
Does the specification extend the given interface?
Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
```

```
>>> I2.extends(I2, strict=False)
1
```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with 'tag'.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```
>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod `isEqualOrExtendedBy` (*other*)

Same interface or extends?

static `isOrExtends` ()

Test whether a specification is or extends another

classmethod `names` (*all=False*)

Return the attribute names defined by the interface.

classmethod `namesAndDescriptions` (*all=False*)

Return attribute names and descriptions defined by interface.

static `providedBy` ()

Test whether an interface is implemented by the specification

classmethod `queryDescriptionFor` (*name, default=None*)

classmethod `queryTaggedValue` (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod `setTaggedValue` (*tag, value*)

Associates 'value' with 'key'.

classmethod `subscribe` (*dependent*)

classmethod `unsubscribe` (*dependent*)

classmethod `validateInvariants` (*obj, errors=None*)

validate object to defined invariants.

classmethod `weakref` (*callback=None*)

ITaskParallelDecorator

```
class IPython.kernel.parallelfunction.ITaskParallelDecorator (name,
                                                                bases=(),
                                                                attrs=None,
                                                                __doc__=None,
                                                                __module__=None,
                                                                __module__=None)
```

Bases: `zope.interface.Interface`

A decorator that creates a parallel function.

classmethod `__init__` (*name, bases=(), attrs=None, __doc__=None, __module__=None*)

classmethod `changed` (*originally_changed*)

We, or something we depend on, have changed

classmethod `deferred` ()

Return a deferred class corresponding to the interface.

classmethod `direct` (*name*)

classmethod `extends` (*interface, strict=True*)

Does the specification extend the given interface?

Test whether an interface in the specification extends the given interface

Examples:

```
>>> from zope.interface import Interface
>>> from zope.interface.declarations import Declaration
>>> class I1(Interface): pass
...
>>> class I2(I1): pass
...
>>> class I3(Interface): pass
...
>>> class I4(I3): pass
```

```

...
>>> spec = Declaration()
>>> int(spec.extends(Interface))
1
>>> spec = Declaration(I2)
>>> int(spec.extends(Interface))
1
>>> int(spec.extends(I1))
1
>>> int(spec.extends(I2))
1
>>> int(spec.extends(I3))
0
>>> int(spec.extends(I4))
0
>>> I2.extends(I2)
0
>>> I2.extends(I2, False)
1
>>> I2.extends(I2, strict=False)
1

```

classmethod `get` (*name*, *default=None*)

Query for an attribute description

classmethod `getBases` ()

classmethod `getDescriptionFor` (*name*)

Return the attribute description for the given name.

classmethod `getDoc` ()

Returns the documentation for the object.

classmethod `getName` ()

Returns the name of the object.

classmethod `getTaggedValue` (*tag*)

Returns the value associated with ‘tag’.

classmethod `getTaggedValueTags` ()

Returns a list of all tags.

static `implementedBy` ()

Test whether the specification is implemented by a class or factory. Raise `TypeError` if argument is neither a class nor a callable.

classmethod `interfaces` ()

Return an iterator for the interfaces in the specification

for example:

```

>>> from zope.interface import Interface
>>> class I1(Interface): pass
...
>>>

```

```
>>> i = I1.interfaces()
>>> i.next().getName()
'I1'
>>> list(i)
[]
```

classmethod **isEqualOrExtendedBy** (*other*)

Same interface or extends?

static **isOrExtends** ()

Test whether a specification is or extends another

classmethod **names** (*all=False*)

Return the attribute names defined by the interface.

classmethod **namesAndDescriptions** (*all=False*)

Return attribute names and descriptions defined by interface.

static **providedBy** ()

Test whether an interface is implemented by the specification

classmethod **queryDescriptionFor** (*name, default=None*)

classmethod **queryTaggedValue** (*tag, default=None*)

Returns the value associated with 'tag'.

classmethod **setTaggedValue** (*tag, value*)

Associates 'value' with 'key'.

classmethod **subscribe** (*dependent*)

classmethod **unsubscribe** (*dependent*)

classmethod **validateInvariants** (*obj, errors=None*)

validate object to defined invariants.

classmethod **weakref** (*callback=None*)

ParallelFunction

class IPython.kernel.parallelfunction.**ParallelFunction** (*mapper*)

Bases: object

The implementation of a parallel function.

A parallel function is similar to Python's map function:

map(func, *sequences) -> pfunc(*sequences)

Parallel functions should be created by using the @parallel decorator.

__init__ (*mapper*)

Create a parallel function from an *IMapper*.

Parameters

mapper [an *IMapper* implementer.] The mapper to use for the parallel function

8.70 kernel.pbutil

8.70.1 Module: `kernel.pbutil`

Utilities for PB using modules.

8.70.2 Functions

`IPython.kernel.pbutil.checkMessageSize(m, info)`

Check string *m* to see if it violates `banana.SIZE_LIMIT`.

This should be used on the client side of things for `push`, `scatter` and `push_serialized` and on the other end for `pull`, `gather` and `pull_serialized`.

Parameters

m [string] Message whose size will be checked.

info [string] String describing what object the message refers to.

Exceptions

- *PBMessageSizeError*: Raised in the message is $>$ `banana.SIZE_LIMIT`

Returns The original message or a Failure wrapping a `PBMessageSizeError`

`IPython.kernel.pbutil.packageFailure(f)`

Clean and pickle a failure preappending the string `FAILURE`:

`IPython.kernel.pbutil.unpackFailure(r)`

See if a returned value is a pickled Failure object.

To distinguish between general pickled objects and pickled Failures, the other side should prepend the string `FAILURE`: to any pickled Failure.

8.71 kernel.pendingdeferred

8.71.1 Module: `kernel.pendingdeferred`

Inheritance diagram for `IPython.kernel.pendingdeferred`:

kernel.pendingdeferred.PendingDeferredManager

Classes to manage pending Deferreds.

A pending deferred is a deferred that may or may not have fired. This module is useful for taking a class whose methods return deferreds and wrapping it to provide API that keeps track of those deferreds for later retrieval. See the tests for examples of its usage.

8.71.2 PendingDeferredManager

class IPython.kernel.pendingdeferred.**PendingDeferredManager**

Bases: object

A class to track pending deferreds.

To track a pending deferred, the user of this class must first get a deferredID by calling *get_next_deferred_id*. Then the user calls *save_pending_deferred* passing that id and the deferred to be tracked. To later retrieve it, the user calls *get_pending_deferred* passing the id.

__init__ ()

Manage pending deferreds.

clear_pending_deferreds ()

Remove all the deferreds I am tracking.

delete_pending_deferred (deferred_id)

Remove a deferred I am tracking and add a null Errback.

Parameters

deferredID [str] The id of a deferred that I am tracking.

get_deferred_id ()

get_pending_deferred (deferred_id, block)

quick_has_id (deferred_id)

save_pending_deferred (d, deferred_id=None)

Save the result of a deferred for later retrieval.

This works even if the deferred has not fired.

Only callbacks and errbacks applied to d before this method is called will be called no the final result.

IPython.kernel.pendingdeferred.**two_phase** (wrapped_method)

Wrap methods that return a deferred into a two phase process.

This transforms:

```
foo(arg1, arg2, ...) -> foo(arg1, arg2,...,block=True).
```

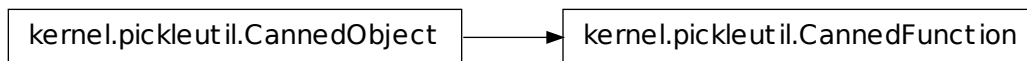
The wrapped method will then return a deferred to a deferred id. This will only work on method of classes that inherit from *PendingDeferredManager*, as that class provides an API for

block is a boolean to determine if we should use the two phase process or just simply call the wrapped method. At this point block does not have a default and it probably won't.

8.72 kernel.pickleutil

8.72.1 Module: kernel.pickleutil

Inheritance diagram for IPython.kernel.pickleutil:



Pickle related utilities.

8.72.2 Classes

CannedFunction

```
class IPython.kernel.pickleutil.CannedFunction(f)
    Bases: IPython.kernel.pickleutil.CannedObject
    __init__(f)
    getFunction(g=None)
```

CannedObject

```
class IPython.kernel.pickleutil.CannedObject
    Bases: object
    __init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

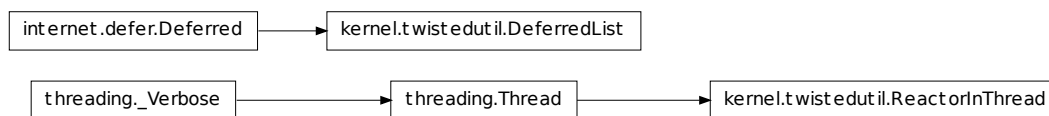
8.72.3 Functions

```
IPython.kernel.pickleutil.can(obj)
IPython.kernel.pickleutil.canDict(obj)
IPython.kernel.pickleutil.canSequence(obj)
IPython.kernel.pickleutil.rebindFunctionGlobals(f, gbls)
IPython.kernel.pickleutil.uncan(obj, g=None)
IPython.kernel.pickleutil.uncanDict(obj, g=None)
IPython.kernel.pickleutil.uncanSequence(obj, g=None)
```

8.73 kernel.twistedutil

8.73.1 Module: kernel.twistedutil

Inheritance diagram for `IPython.kernel.twistedutil`:



Things directly related to all of twisted.

8.73.2 Classes

DeferredList

```
class IPython.kernel.twistedutil.DeferredList(deferredList, fireOnOneCall-
                                             back=0, fireOnOneErrback=0,
                                             consumeErrors=0, logErrors=0)
```

Bases: `twisted.internet.defer.Deferred`

I combine a group of deferreds into one callback.

I track a list of `L{Deferred}`s for their callbacks, and make a single callback when they have all completed, a list of (success, result) tuples, ‘success’ being a boolean.

Note that you can still use a `L{Deferred}` after putting it in a `DeferredList`. For example, you can suppress ‘Unhandled error in Deferred’ messages by adding errbacks to the `Deferreds` *after* putting them in the `DeferredList`, as a `DeferredList` won’t swallow the errors. (Although a more convenient way to do this is simply to set the `consumeErrors` flag)

Note: This is a modified version of the `twisted.internet.defer.DeferredList`

__init__ (*deferredList*, *fireOnOneCallback*=0, *fireOnOneErrback*=0, *consumeErrors*=0, *logErrors*=0)

Initialize a DeferredList.

@type deferredList: C{list} of L{Deferred}s @param deferredList: The list of deferreds to track. @param fireOnOneCallback: (keyword param) a flag indicating that

only one callback needs to be fired for me to call my callback

@param fireOnOneErrback: (keyword param) a flag indicating that only one errback needs to be fired for me to call my errback

@param consumeErrors: (keyword param) a flag indicating that any errors raised in the original deferreds should be consumed by this DeferredList. This is useful to prevent spurious warnings being logged.

addBoth (*callback*, **args*, ***kw*)

Convenience method for adding a single callable as both a callback and an errback.

See L{addCallbacks}.

addCallback (*callback*, **args*, ***kw*)

Convenience method for adding just a callback.

See L{addCallbacks}.

addCallbacks (*callback*, *errback*=None, *callbackArgs*=None, *callbackKeywords*=None, *errbackArgs*=None, *errbackKeywords*=None)

Add a pair of callbacks (success and error) to this Deferred.

These will be executed when the ‘master’ callback is run.

addErrback (*errback*, **args*, ***kw*)

Convenience method for adding just an errback.

See L{addCallbacks}.

callback (*result*)

Run all success callbacks that have been added to this Deferred.

Each callback will have its result passed as the first argument to the next; this way, the callbacks act as a ‘processing chain’. Also, if the success-callback returns a Failure or raises an Exception, processing will continue on the *error*-callback chain.

chainDeferred (*d*)

Chain another Deferred to this Deferred.

This method adds callbacks to this Deferred to call d’s callback or errback, as appropriate. It is merely a shorthand way of performing the following:

```
self.addCallbacks(d.callback, d.errback)
```

When you chain a deferred d2 to another deferred d1 with `d1.chainDeferred(d2)`, you are making d2 participate in the callback chain of d1. Thus any event that fires d1 will also fire d2. However, the converse is B{not} true; if d2 is fired d1 will not be affected.

errback (*fail=None*)

Run all error callbacks that have been added to this Deferred.

Each callback will have its result passed as the first argument to the next; this way, the callbacks act as a ‘processing chain’. Also, if the error-callback returns a non-Failure or doesn’t raise an Exception, processing will continue on the *success*-callback chain.

If the argument that’s passed to me is not a failure.Failure instance, it will be embedded in one. If no argument is passed, a failure.Failure instance will be created based on the current traceback stack.

Passing a string as ‘fail’ is deprecated, and will be punished with a warning message.

@raise NoCurrentExceptionError: If C{fail} is C{None} but there is no current exception state.

pause ()

Stop processing on a Deferred until `L{unpause}()` is called.

setTimeout (*seconds*, *timeoutFunc*=<function timeout at 0x107f1b938>, *args, **kw)

Set a timeout function to be triggered if I am not called.

@param seconds: How long to wait (from now) before firing the timeoutFunc.

@param timeoutFunc: will receive the Deferred and *args, **kw as its arguments. The default timeoutFunc will call the errback with a `L{TimeoutError}`.

unpause ()

Process all callbacks made since `L{pause}()` was called.

ReactorInThread

```
class IPython.kernel.twistedutil.ReactorInThread (group=None,    target=None,
                                                  name=None,      args=(),
                                                  kwargs=None,    ver-
                                                 bose=None)
```

Bases: `threading.Thread`

Run the twisted reactor in a different thread.

For the process to be able to exit cleanly, do the following:

```
rit = ReactorInThread() rit.setDaemon(True) rit.start()
```

```
__init__ (group=None, target=None, name=None, args=(), kwargs=None, verbose=None)
```

daemon

getName ()

ident

isAlive()**isDaemon()****is_alive()****join**(*timeout=None*)**name****run()**

Run the twisted reactor in a thread.

This runs the reactor with `installSignalHandlers=0`, which prevents twisted from installing any of its own signal handlers. This needs to be disabled because `signal.signal` can't be called in a thread. The only problem with this is that `SIGCHLD` events won't be detected so `spawnProcess` won't detect that its processes have been killed by an external factor.

setDaemon(*daemonic*)**setName**(*name*)**start()****stop()**

8.73.3 Functions

`IPython.kernel.twistedutil.gatherBoth`(*dlist*, *fireOnOneCallback=0*, *fireOnOneErrback=0*, *consumeErrors=0*, *logErrors=0*)

This is like `gatherBoth`, but sets `consumeErrors=1`.

`IPython.kernel.twistedutil.make_deferred`(*func*)

A decorator that calls a function with `:func'maybeDeferred'`.

`IPython.kernel.twistedutil.parseResults`(*results*)

Pull out results/Failures from a `DeferredList`.

`IPython.kernel.twistedutil.sleep_deferred`(*seconds*)

Sleep without blocking the event loop.

`IPython.kernel.twistedutil.wait_for_file`(*filename*, *delay=0.10000000000000001*, *max_tries=10*)

Wait (poll) for a file to be created.

This method returns a `Deferred` that will fire when a file exists. It works by polling `os.path.isfile` in time intervals specified by the `delay` argument. If `max_tries` is reached, it will `errback` with a `FileTimeoutError`.

Parameters **filename** : str

The name of the file to wait for.

delay : float

The time to wait between polls.

max_tries : int

The max number of attempts before raising *FileTimeoutError*

Returns **d** : Deferred

A Deferred instance that will fire when the file exists.

8.74 kernel.util

8.74.1 Module: `kernel.util`

General utilities for kernel related things.

8.74.2 Functions

`IPython.kernel.util.catcher(r)`

`IPython.kernel.util.curry(f, *curryArgs, **curryKWargs)`

Curry the function *f* with *curryArgs* and *curryKWargs*.

`IPython.kernel.util.printer(r, msg='')`

`IPython.kernel.util.tarModule(mod)`

Makes a tarball (as a string) of a locally imported module.

This method looks at the `__file__` attribute of an imported module and makes a tarball of the top level of the module. It then reads the tarball into a binary string.

The method returns the tarball's name and the binary string representing the tarball.

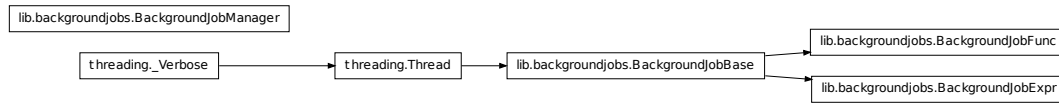
Notes:

- It will handle both single module files, as well as packages.
- The byte code files (*.pyc) are not deleted.
- It has not been tested with modules containing extension code, but it should work in most cases.
- There are cross platform issues.

8.75 lib.backgroundjobs

8.75.1 Module: `lib.backgroundjobs`

Inheritance diagram for `IPython.lib.backgroundjobs`:



Manage background (threaded) jobs conveniently from an interactive shell.

This module provides a `BackgroundJobManager` class. This is the main class meant for public usage, it implements an object which can create and manage new background jobs.

It also provides the actual job classes managed by these `BackgroundJobManager` objects, see their docstrings below.

This system was inspired by discussions with B. Granger and the `BackgroundCommand` class described in the book *Python Scripting for Computational Science*, by H. P. Langtangen:

<http://folk.uio.no/hpl/scripting>

(although ultimately no code from this text was used, as IPython's system is a separate implementation).

8.75.2 Classes

BackgroundJobBase

class IPython.lib.backgroundjobs.**BackgroundJobBase**

Bases: `threading.Thread`

Base class to build `BackgroundJob` classes.

The derived classes must implement:

- Their own `__init__`, since the one here raises `NotImplementedError`. The derived constructor must call `self._init()` at the end, to provide common initialization.
- A `str`form attribute used in calls to `__str__`.
- A `call()` method, which will make the actual execution call and must return a value to be held in the 'result' field of the job object.

`__init__()`

`daemon`

`getName()`

`ident`

`isAlive()`

`isDaemon()`

```
is_alive()  
join(timeout=None)  
name  
run()  
setDaemon(daemonic)  
setName(name)  
start()  
traceback()
```

BackgroundJobExpr

```
class IPython.lib.backgroundjobs.BackgroundJobExpr(expression, glob=None,  
                                                    loc=None)
```

Bases: `IPython.lib.backgroundjobs.BackgroundJobBase`

Evaluate an expression as a background job (uses a separate thread).

```
__init__(expression, glob=None, loc=None)
```

Create a new job from a string which can be fed to `eval()`.

global/locals dicts can be provided, which will be passed to the `eval` call.

```
call()  
daemon  
getName()  
ident  
isAlive()  
isDaemon()  
is_alive()  
join(timeout=None)  
name  
run()  
setDaemon(daemonic)  
setName(name)  
start()  
traceback()
```

BackgroundJobFunc

class IPython.lib.backgroundjobs.**BackgroundJobFunc** (*func, *args, **kwargs*)

Bases: IPython.lib.backgroundjobs.**BackgroundJobBase**

Run a function call as a background job (uses a separate thread).

___**init**___ (*func, *args, **kwargs*)

Create a new job from a callable object.

Any positional arguments and keyword args given to this constructor after the initial callable are passed directly to it.

call ()

daemon

getName ()

ident

isAlive ()

isDaemon ()

is_alive ()

join (*timeout=None*)

name

run ()

setDaemon (*daemonic*)

setName (*name*)

start ()

traceback ()

BackgroundJobManager

class IPython.lib.backgroundjobs.**BackgroundJobManager**

Class to manage a pool of backgrounded threaded jobs.

Below, we assume that 'jobs' is a BackgroundJobManager instance.

Usage summary (see the method docstrings for details):

jobs.new(...) -> start a new job

jobs() or jobs.status() -> print status summary of all jobs

jobs[N] -> returns job number N.

foo = jobs[N].result -> assign to variable foo the result of job N

jobs[N].traceback() -> print the traceback of dead job N

`jobs.remove(N)` -> remove (finished) job N

`jobs.flush_finished()` -> remove all finished jobs

As a convenience feature, `BackgroundJobManager` instances provide the utility result and traceback methods which retrieve the corresponding information from the jobs list:

`jobs.result(N) <-> jobs[N].result` `jobs.traceback(N) <-> jobs[N].traceback()`

While this appears minor, it allows you to use tab completion interactively on the job manager instance.

In interactive mode, IPython provides the magic fuction `%bg` for quick creation of backgrounded expression-based jobs. Type `bg?` for details.

`__init__()`

`flush_finished()`

Flush all jobs finished (completed and dead) from lists.

Running jobs are never flushed.

It first calls `_status_new()`, to update info. If any jobs have completed since the last `_status_new()` call, the flush operation aborts.

`new(func_or_exp, *args, **kwargs)`

Add a new background job and start it in a separate thread.

There are two types of jobs which can be created:

1. Jobs based on expressions which can be passed to an `eval()` call. The expression must be given as a string. For example:

```
job_manager.new('myfunc(x,y,z=1)',glob[,loc])
```

The given expression is passed to `eval()`, along with the optional global/local dicts provided. If no dicts are given, they are extracted automatically from the caller's frame.

A Python statement is NOT a valid `eval()` expression. Basically, you can only use as an `eval()` argument something which can go on the right of an '=' sign and be assigned to a variable.

For example, `"print 'hello'"` is not valid, but `'2+3'` is.

2. Jobs given a function object, optionally passing additional positional arguments:

```
job_manager.new(myfunc,x,y)
```

The function is called with the given arguments.

If you need to pass keyword arguments to your function, you must supply them as a dict named `kw`:

```
job_manager.new(myfunc,x,y,kw=dict(z=1))
```

The reason for this asymmetry is that the `new()` method needs to maintain access to its own keywords, and this prevents name collisions between arguments to `new()` and arguments to your own functions.

In both cases, the result is stored in the `job.result` field of the background job object.

Notes and caveats:

1. All threads running share the same standard output. Thus, if your background jobs generate output, it will come out on top of whatever you are currently writing. For this reason, background jobs are best used with silent functions which simply return their output.
2. Threads also all work within the same global namespace, and this system does not lock interactive variables. So if you send job to the background which operates on a mutable object for a long time, and start modifying that same mutable object interactively (or in another backgrounded job), all sorts of bizarre behaviour will occur.
3. If a background job is spending a lot of time inside a C extension module which does not release the Python Global Interpreter Lock (GIL), this will block the IPython prompt. This is simply because the Python interpreter can only switch between threads at Python bytecodes. While the execution is inside C code, the interpreter must simply wait unless the extension module releases the GIL.
4. There is no way, due to limitations in the Python threads library, to kill a thread once it has started.

remove (*num*)

Remove a finished (completed or dead) job.

result (*num*)

result(N) -> return the result of job N.

status (*verbose=0*)

Print a status of all jobs currently being managed.

traceback (*num*)

8.76 lib.clipboard

8.76.1 Module: lib.clipboard

Utilities for accessing the platform's clipboard.

8.76.2 Functions

`IPython.lib.clipboard.osx_clipboard_get()`

Get the clipboard's text on OS X.

`IPython.lib.clipboard.tkinter_clipboard_get()`

Get the clipboard's text using Tkinter.

This is the default on systems that are not Windows or OS X. It may interfere with other UI toolkits and should be replaced with an implementation that uses that toolkit.

`IPython.lib.clipboard.win32_clipboard_get()`

Get the current clipboard's text on Windows.

Requires Mark Hammond's pywin32 extensions.

8.77 lib.deepreload

8.77.1 Module: lib.deepreload

A module to change reload() so that it acts recursively. To enable it type:

```
import __builtin__, deepreload
__builtin__.reload = deepreload.reload
```

You can then disable it with:

```
__builtin__.reload = deepreload.original_reload
```

Alternatively, you can add a dreload builtin alongside normal reload with:

```
__builtin__.dreload = deepreload.reload
```

This code is almost entirely based on knee.py from the standard library.

8.77.2 Functions

```
IPython.lib.deepreload.deep_import_hook(name, globals=None, locals=None,
                                          fromlist=None, level=-1)
```

```
IPython.lib.deepreload.deep_reload_hook(module)
```

```
IPython.lib.deepreload.determine_parent(globals)
```

```
IPython.lib.deepreload.ensure_fromlist(m, fromlist, recursive=0)
```

```
IPython.lib.deepreload.find_head_package(parent, name)
```

```
IPython.lib.deepreload.import_module(partname, fqname, parent)
```

```
IPython.lib.deepreload.load_tail(q, tail)
```

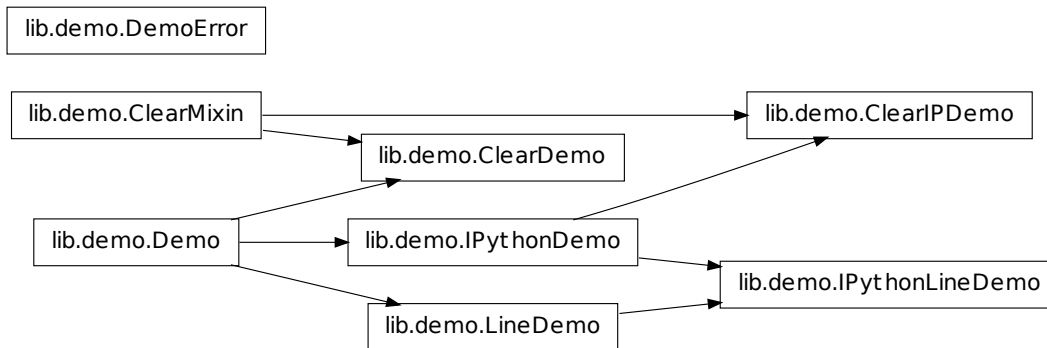
```
IPython.lib.deepreload.reload(module, exclude=['sys', '__builtin__', '__main__'])
```

Recursively reload all modules used in the given module. Optionally takes a list of modules to exclude from reloading. The default exclude list contains sys, __main__, and __builtin__, to prevent, e.g., resetting display, exception, and io hooks.

8.78 lib.demo

8.78.1 Module: lib.demo

Inheritance diagram for IPython.lib.demo:



Module for interactive demos using IPython.

This module implements a few classes for running Python scripts interactively in IPython for demonstrations. With very simple markup (a few tags in comments), you can control points where the script stops executing and returns control to IPython.

Provided classes

The classes are (see their docstrings for further details):

- `Demo`: pure python demos
- `IPythonDemo`: demos with input to be processed by IPython as if it had been typed interactively (so magics work, as well as any other special syntax you may have added via input prefilters).
- `LineDemo`: single-line version of the `Demo` class. These demos are executed one line at a time, and require no markup.
- `IPythonLineDemo`: IPython version of the `LineDemo` class (the demo is executed a line at a time, but processed via IPython).
- `ClearMixin`: mixin to make `Demo` classes with less visual clutter. It declares an empty `marquee` and a `pre_cmd` that clears the screen before each block (see Subclassing below).
- `ClearDemo`, `ClearIPDemo`: mixin-enabled versions of the `Demo` and `IPythonDemo` classes.

Subclassing

The classes here all include a few methods meant to make customization by subclassing more convenient. Their docstrings below have some more details:

- `marquee()`: generates a marquee to provide visible on-screen markers at each block start and end.
- `pre_cmd()`: run right before the execution of each block.
- `post_cmd()`: run right after the execution of each block. If the block raises an exception, this is NOT called.

Operation

The file is run in its own empty namespace (though you can pass it a string of arguments as if in a command line environment, and it will see those as `sys.argv`). But at each stop, the global IPython namespace is updated with the current internal demo namespace, so you can work interactively with the data accumulated so far.

By default, each block of code is printed (with syntax highlighting) before executing it and you have to confirm execution. This is intended to show the code to an audience first so you can discuss it, and only proceed with execution once you agree. There are a few tags which allow you to modify this behavior.

The supported tags are:

`# <demo> stop`

Defines block boundaries, the points where IPython stops execution of the file and returns to the interactive prompt.

You can optionally mark the stop tag with extra dashes before and after the word ‘stop’, to help visually distinguish the blocks in a text editor:

`# <demo> — stop —`

`# <demo> silent`

Make a block execute silently (and hence automatically). Typically used in cases where you have some boilerplate or initialization code which you need executed but do not want to be seen in the demo.

`# <demo> auto`

Make a block execute automatically, but still being printed. Useful for simple code which does not warrant discussion, since it avoids the extra manual confirmation.

`# <demo> auto_all`

This tag can `_only_` be in the first block, and if given it overrides the individual auto tags to make the whole demo fully automatic (no block asks for confirmation). It can also be given at creation time (or the attribute set later) to override what’s in the file.

While `_any_` python file can be run as a Demo instance, if there are no stop tags the whole file will run in a single block (no different that calling first `%pycat` and then `%run`). The minimal markup to make this useful is to place a set of stop tags; the other tags are only there to let you fine-tune the execution.

This is probably best explained with the simple example file below. You can copy this into a file named `ex_demo.py`, and try running it via:

from IPython.demo import Demo d = Demo('ex_demo.py') d() <— Call the d object (omit the parens if you have autocall set to 2).

Each time you call the demo object, it runs the next block. The demo object has a few useful methods for navigation, like again(), edit(), jump(), seek() and back(). It can be reset for a new run via reset() or reloaded from disk (in case you’ve edited the source) via reload(). See their docstrings below.

Note: To make this simpler to explore, a file called “demo-exercizer.py” has been added to the “docs/examples/core” directory. Just cd to this directory in an IPython session, and type:

```
%run demo-exercizer.py
```

and then follow the directions.

Example

The following is a very simple example of a valid demo file.

```
##### EXAMPLE DEMO <ex_demo.py> ##### '''A
simple interactive demo to illustrate the use of IPython's Demo class.'''

print 'Hello, welcome to an interactive IPython demo.'

# The mark below defines a block boundary, which is a point where IPython will # stop execution and return
to the interactive prompt. The dashes are actually # optional and used only as a visual aid to clearly separate
blocks while # editing the demo code. # <demo> stop

x = 1 y = 2

# <demo> stop

# the mark below makes this block as silent # <demo> silent

print 'This is a silent block, which gets executed but not printed.'

# <demo> stop # <demo> auto print 'This is an automatic block.' print 'It is executed without asking for
confirmation, but printed.' z = x+y

print 'z=',x

# <demo> stop # This is just another normal block. print 'z is now:', z

print 'bye!' ##### END EXAMPLE DEMO <ex_demo.py>
#####
```

8.78.2 Classes

ClearDemo

```
class IPython.lib.demo.ClearDemo(src, title='', arg_str='', auto_all=None)
    Bases: IPython.lib.demo.ClearMixin, IPython.lib.demo.Demo
```

__init__ (*src*, *title*='', *arg_str*='', *auto_all*=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use `IPython.Demo?` in IPython to see it).

Inputs:

- **src** is either a file, or file-like object, or a string that can be resolved to a filename.

Optional inputs:

- **title**: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

- **arg_str**(''): a string of arguments, internally converted to a list

just like `sys.argv`, so the demo script can see a similar environment.

- **auto_all**(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again ()

Move the seek pointer back one block and re-execute.

back (*num*=1)

Move the seek pointer back *num* blocks (default is 1).

edit (*index*=None)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use `reload()` when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload ()

Load file object.

jump (*num*=1)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee (*txt*='', *width*=78, *mark*='*')

Blank marquee that returns '' no matter what the input.

post_cmd ()

Method called after executing each block.

pre_cmd()

Method called before executing each block.

This one simply clears the screen.

reload()

Reload source from disk and initialize state.

reset()

Reset the namespace and seek pointer to restart the demo

run_cell(source)

Execute a string with one or more lines of code

seek(index)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show(index=None)

Show a single block on screen

show_all()

Show entire demo on screen, block by block

ClearIPDemo

class IPython.lib.demo.**ClearIPDemo**(src, title='', arg_str='', auto_all=None)

Bases: IPython.lib.demo.ClearMixin, IPython.lib.demo.IPythonDemo

__init__(src, title='', arg_str='', auto_all=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

- src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

- title:** a string to use as the demo name. Of most use when the demo you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.
- arg_str('')**: a string of arguments, internally converted to a list just like sys.argv, so the demo script can see a similar environment.
- auto_all(None):** global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again()

Move the seek pointer back one block and re-execute.

back (*num=1*)

Move the seek pointer back num blocks (default is 1).

edit (*index=None*)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use `reload()` when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload ()

Load file object.

jump (*num=1*)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee (*txt='', width=78, mark='*'*)

Blank marquee that returns '' no matter what the input.

post_cmd ()

Method called after executing each block.

pre_cmd ()

Method called before executing each block.

This one simply clears the screen.

reload ()

Reload source from disk and initialize state.

reset ()

Reset the namespace and seek pointer to restart the demo

run_cell (*source*)

Execute a string with one or more lines of code

seek (*index*)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show (*index=None*)

Show a single block on screen

show_all()

Show entire demo on screen, block by block

ClearMixin

class IPython.lib.demo.**ClearMixin**

Bases: object

Use this mixin to make Demo classes with less visual clutter.

Demos using this mixin will clear the screen before every block and use blank marquees.

Note that in order for the methods defined here to actually override those of the classes it's mixed with, it must go /first/ in the inheritance tree. For example:

```
class ClearIPDemo(ClearMixin,IPythonDemo): pass
```

will provide an IPythonDemo class with the mixin's features.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

marquee (txt='', width=78, mark='*')

Blank marquee that returns '' no matter what the input.

pre_cmd()

Method called before executing each block.

This one simply clears the screen.

Demo

class IPython.lib.demo.**Demo** (src, title='', arg_str='', auto_all=None)

Bases: object

__init__ (src, title='', arg_str='', auto_all=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

- src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

- title:** a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

- arg_str('')**: a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

- `auto_all(None)`: global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again()

Move the seek pointer back one block and re-execute.

back(*num=1*)

Move the seek pointer back *num* blocks (default is 1).

edit(*index=None*)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use `reload()` when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload()

Load file object.

jump(*num=1*)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee(*txt=' ', width=78, mark='*'*)

Return the input string centered in a 'marquee'.

post_cmd()

Method called after executing each block.

pre_cmd()

Method called before executing each block.

reload()

Reload source from disk and initialize state.

reset()

Reset the namespace and seek pointer to restart the demo

run_cell(*source*)

Execute a string with one or more lines of code

seek(*index*)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show(*index=None*)

Show a single block on screen

show_all()

Show entire demo on screen, block by block

DemoError

class IPython.lib.demo.DemoError

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

IPythonDemo

class IPython.lib.demo.IPythonDemo (src, title='', arg_str='', auto_all=None)

Bases: IPython.lib.demo.Demo

Class for interactive demos with IPython's input processing applied.

This subclasses Demo, but instead of executing each block by the Python interpreter (via exec), it actually calls IPython on it, so that any input filters which may be in place are applied to the input block.

If you have an interactive environment which exposes special input processing, you can use this class instead to write demo scripts which operate exactly as if you had typed them interactively. The default Demo class requires the input to be valid, pure Python code.

__init__ (src, title='', arg_str='', auto_all=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

- src is either a file, or file-like object, or a string** that can be resolved to a filename.

Optional inputs:

- title**: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

- arg_str('')**: a string of arguments, internally converted to a list

just like sys.argv, so the demo script can see a similar environment.

- auto_all(None)**: global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again()

Move the seek pointer back one block and re-execute.

back (*num=1*)

Move the seek pointer back num blocks (default is 1).

edit (*index=None*)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use `reload()` when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload ()

Load file object.

jump (*num=1*)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee (*txt=' ', width=78, mark='*'*)

Return the input string centered in a 'marquee'.

post_cmd ()

Method called after executing each block.

pre_cmd ()

Method called before executing each block.

reload ()

Reload source from disk and initialize state.

reset ()

Reset the namespace and seek pointer to restart the demo

run_cell (*source*)

Execute a string with one or more lines of code

seek (*index*)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show (*index=None*)

Show a single block on screen

show_all ()

Show entire demo on screen, block by block

IPythonLineDemo

class IPython.lib.demo.**IPythonLineDemo** (*src*, *title*='', *arg_str*='', *auto_all*=None)

Bases: IPython.lib.demo.IPythonDemo, IPython.lib.demo.LineDemo

Variant of the LineDemo class whose input is processed by IPython.

__init__ (*src*, *title*='', *arg_str*='', *auto_all*=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

- src is either a file, or file-like object, or a** string that can be resolved to a filename.

Optional inputs:

- title:** a string to use as the demo name. Of most use when the demo you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.
- arg_str('')**: a string of arguments, internally converted to a list just like sys.argv, so the demo script can see a similar environment.
- auto_all(None):** global flag to run all blocks automatically without confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again ()

Move the seek pointer back one block and re-execute.

back (*num*=1)

Move the seek pointer back num blocks (default is 1).

edit (*index*=None)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use reload() when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload ()

Load file object.

jump (*num*=1)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee (*txt*='', *width*=78, *mark*='*')

Return the input string centered in a 'marquee'.

post_cmd ()

Method called after executing each block.

pre_cmd ()

Method called before executing each block.

reload ()

Reload source from disk and initialize state.

reset ()

Reset the namespace and seek pointer to restart the demo

run_cell (*source*)

Execute a string with one or more lines of code

seek (*index*)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show (*index*=None)

Show a single block on screen

show_all ()

Show entire demo on screen, block by block

LineDemo

class IPython.lib.demo.**LineDemo** (*src*, *title*='', *arg_str*='', *auto_all*=None)

Bases: IPython.lib.demo.Demo

Demo where each line is executed as a separate block.

The input script should be valid Python code.

This class doesn't require any markup at all, and it's meant for simple scripts (with no nesting or any kind of indentation) which consist of multiple lines of input to be executed, one at a time, as if they had been typed in the interactive prompt.

Note: the input can not have *any* indentation, which means that only single-lines of input are accepted, not even function definitions are valid.

__init__ (*src*, *title*='', *arg_str*='', *auto_all*=None)

Make a new demo object. To run the demo, simply call the object.

See the module docstring for full details and an example (you can use IPython.Demo? in IPython to see it).

Inputs:

- src** is either a file, or file-like object, or a string that can be resolved to a filename.

Optional inputs:

- title**: a string to use as the demo name. Of most use when the demo

you are making comes from an object that has no filename, or if you want an alternate denotation distinct from the filename.

- arg_str**(''): a string of arguments, internally converted to a list

just like `sys.argv`, so the demo script can see a similar environment.

- auto_all**(None): global flag to run all blocks automatically without

confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

again()

Move the seek pointer back one block and re-execute.

back(*num=1*)

Move the seek pointer back *num* blocks (default is 1).

edit(*index=None*)

Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use `reload()` when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.

fload()

Load file object.

jump(*num=1*)

Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.

marquee(*txt=' ', width=78, mark='*'*)

Return the input string centered in a 'marquee'.

post_cmd()

Method called after executing each block.

pre_cmd()

Method called before executing each block.

reload()

Reload source from disk and initialize state.

reset()

Reset the namespace and seek pointer to restart the demo

run_cell(*source*)

Execute a string with one or more lines of code

seek (*index*)

Move the current seek pointer to the given block.

You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show (*index=None*)

Show a single block on screen

show_all ()

Show entire demo on screen, block by block

8.78.3 Function

`IPython.lib.demo.re_mark` (*mark*)

8.79 lib.guisupport

8.79.1 Module: lib.guisupport

Support for creating GUI apps and starting event loops.

IPython's GUI integration allows interactive plotting and GUI usage in IPython session. IPython has two different types of GUI integration:

1. The terminal based IPython supports GUI event loops through Python's `PyOS_InputHook`. `PyOS_InputHook` is a hook that Python calls periodically whenever `raw_input` is waiting for a user to type code. We implement GUI support in the terminal by setting `PyOS_InputHook` to a function that iterates the event loop for a short while. It is important to note that in this situation, the real GUI event loop is NOT run in the normal manner, so you can't use the normal means to detect that it is running.
2. In the two process IPython kernel/frontend, the GUI event loop is run in the kernel. In this case, the event loop is run in the normal manner by calling the function or method of the GUI toolkit that starts the event loop.

In addition to starting the GUI event loops in one of these two ways, IPython will *always* create an appropriate GUI application object when GUI integration is enabled.

If you want your GUI apps to run in IPython you need to do two things:

1. Test to see if there is already an existing main application object. If there is, you should use it. If there is not an existing application object you should create one.
2. Test to see if the GUI event loop is running. If it is, you should not start it. If the event loop is not running you may start it.

This module contains functions for each toolkit that perform these things in a consistent manner. Because of how `PyOS_InputHook` runs the event loop you cannot detect if the event loop is running using the traditional calls (such as `wx.GetApp.IsMainLoopRunning()` in `wxPython`). If `PyOS_InputHook` is set These methods will return a false negative. That is, they will say the event loop is not running, when is actually is. To work around this limitation we proposed the following informal protocol:

- Whenever someone starts the event loop, they *must* set the `_in_event_loop` attribute of the main application object to `True`. This should be done regardless of how the event loop is actually run.
- Whenever someone stops the event loop, they *must* set the `_in_event_loop` attribute of the main application object to `False`.
- If you want to see if the event loop is running, you *must* use `hasattr` to see if `_in_event_loop` attribute has been set. If it is set, you *must* use its value. If it has not been set, you can query the toolkit in the normal manner.
- If you want GUI support and no one else has created an application or started the event loop you *must* do this. We don't want projects to attempt to defer these things to someone else if they themselves need it.

The functions below implement this logic for each GUI toolkit. If you need to create custom application subclasses, you will likely have to modify this code for your own purposes. This code can be copied into your own project so you don't have to depend on IPython.

8.79.2 Functions

`IPython.lib.guisupport.get_app_qt4(*args, **kwargs)`
Create a new qt4 app or return an existing one.

`IPython.lib.guisupport.get_app_wx(*args, **kwargs)`
Create a new wx app or return an exiting one.

`IPython.lib.guisupport.is_event_loop_running_qt4(app=None)`
Is the qt4 event loop running.

`IPython.lib.guisupport.is_event_loop_running_wx(app=None)`
Is the wx event loop running.

`IPython.lib.guisupport.start_event_loop_qt4(app=None)`
Start the qt4 event loop in a consistent manner.

`IPython.lib.guisupport.start_event_loop_wx(app=None)`
Start the wx event loop in a consistent manner.

8.80 lib.inputhook

8.80.1 Module: lib.inputhook

Inheritance diagram for `IPython.lib.inputhook`:

lib.inputhook.InputHookManager

Inputhook management for GUI event loop integration.

8.80.2 InputHookManager

class IPython.lib.inputhook.**InputHookManager**

Bases: object

Manage PyOS_InputHook for different GUI toolkits.

This class installs various hooks under PyOSInputHook to handle GUI event loop integration.

__init__()

clear_app_refs (*gui=None*)

Clear IPython's internal reference to an application instance.

Whenever we create an app for a user on qt4 or wx, we hold a reference to the app. This is needed because in some cases bad things can happen if a user doesn't hold a reference themselves. This method is provided to clear the references we are holding.

Parameters **gui** : None or str

If None, clear all app references. If ('wx', 'qt4') clear the app for that toolkit. References are not held for gtk or tk as those toolkits don't have the notion of an app.

clear_inputhook (*app=None*)

Set PyOS_InputHook to NULL and return the previous one.

Parameters **app** : optional, ignored

This parameter is allowed only so that clear_inputhook() can be called with a similar interface as all the enable_* methods. But the actual value of the parameter is ignored. This uniform interface makes it easier to have user-level entry points in the main IPython app like `enable_gui()`.

current_gui()

Return a string indicating the currently active GUI or None.

disable_gtk()

Disable event loop integration with PyGTK.

This merely sets PyOS_InputHook to NULL.

disable_qt4()

Disable event loop integration with PyQt4.

This merely sets `PyOS_InputHook` to `NULL`.

`disable_tk()`

Disable event loop integration with Tkinter.

This merely sets `PyOS_InputHook` to `NULL`.

`disable_wx()`

Disable event loop integration with wxPython.

This merely sets `PyOS_InputHook` to `NULL`.

`enable_gtk(app=False)`

Enable event loop integration with PyGTK.

Parameters `app` : bool

Create a running application object or not. Because `gtk` doesn't have an `app` class, this does nothing.

Notes

This method sets the `PyOS_InputHook` for PyGTK, which allows the PyGTK to integrate with terminal based applications like IPython.

`enable_qt4()`

Enable event loop integration with PyQt4.

Parameters `app` : bool

Create a running application object or not.

Notes

This method sets the `PyOS_InputHook` for PyQt4, which allows the PyQt4 to integrate with terminal based applications like IPython.

If `app` is `True`, we create an `QApplication` as follows:

```
from PyQt4 import QtCore
app = QtGui.QApplication(sys.argv)
```

But, we first check to see if an application has already been created. If so, we simply return that instance.

`enable_tk(app=False)`

Enable event loop integration with Tk.

Parameters `app` : bool

Create a running application object or not.

Notes

Currently this is a no-op as creating a `Tkinter.Tk` object sets `PyOS_InputHook`.

enable_wx()

Enable event loop integration with wxPython.

Parameters `app` : bool

Create a running application object or not.

Notes

This methods sets the `PyOS_InputHook` for wxPython, which allows the wxPython to integrate with terminal based applications like IPython.

If `app` is `True`, we create an `wx.App` as follows:

```
import wx
app = wx.App(redirect=False, clearSigInt=False)
```

Both options this constructor are important for things to work properly in an interactive context.

But, we first check to see if an application has already been created. If so, we simply return that instance.

get_pyos_inthook()

Return the current `PyOS_InputHook` as a `ctypes.c_void_p`.

get_pyos_inthook_as_func()

Return the current `PyOS_InputHook` as a `ctypes.PYFUNCTYPE`.

set_inthook(callback)

Set `PyOS_InputHook` to `callback` and return the previous one.

`IPython.lib.inthook.enable_gui(gui=None)`

Switch amongst GUI input hooks by name.

This is just a utility wrapper around the methods of the `InputHookManager` object.

Parameters `gui` : optional, string or None

If `None`, clears input hook, otherwise it must be one of the recognized GUI names (see `GUI_*` constants in module).

app : optional, bool

If true, create an app object and return it.

Returns The output of the underlying gui switch routine, typically the actual :

`PyOS_InputHook` wrapper object or the GUI toolkit app created, if there was :

one. :

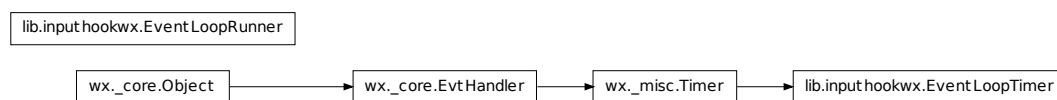
8.81 lib.inputhookgtk

8.81.1 Module: `lib.inputhookgtk`

8.82 lib.inputhookwx

8.82.1 Module: `lib.inputhookwx`

Inheritance diagram for `IPython.lib.inputhookwx`:



Enable wxPython to be used interactive by setting `PyOS_InputHook`.

Authors: Robin Dunn, Brian Granger, Ondrej Certik

8.82.2 Classes

EventLoopRunner

class `IPython.lib.inputhookwx.EventLoopRunner`

Bases: `object`

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`Run(time)`

`check_stdin()`

EventLoopTimer

class `IPython.lib.inputhookwx.EventLoopTimer(func)`

Bases: `wx._misc.Timer`

`__init__(func)`

`AddPendingEvent(self, Event event)`

`Bind(event, handler, source=None, id=-1, id2=-1)`

Bind an event to an event handler.

Parameters

- **event** – One of the EVT_* objects that specifies the type of event to bind,
- **handler** – A callable object to be invoked when the event is delivered to self. Pass None to disconnect an event handler.
- **source** – Sometimes the event originates from a different window than self, but you still want to catch it in self. (For example, a button event delivered to a frame.) By passing the source of the event, the event handling system is able to differentiate between the same event type from different controls.
- **id** – Used to specify the event source by ID instead of instance.
- **id2** – Used when it is desirable to bind a handler to a range of IDs, such as with EVT_MENU_RANGE.

ClassName

See *GetClassName*

Connect (*self*, *int id*, *int lastId*, *EventType eventType*, *PyObject func*)

DeletePendingEvents (*self*)

Destroy ()

NO-OP: Timers must be destroyed by normal reference counting

Disconnect (**args*, ***kwargs*)

Disconnect(*self*, *int id*, *int lastId=-1*, *EventType eventType=wxEVT_NULL*, *PyObject func=None*) -> bool

EvtHandlerEnabled

See *GetEvtHandlerEnabled* and *SetEvtHandlerEnabled*

GetClassName (**args*, ***kwargs*)

GetClassName(*self*) -> String

Returns the class name of the C++ class using wxRTTI.

GetEvtHandlerEnabled (**args*, ***kwargs*)

GetEvtHandlerEnabled(*self*) -> bool

GetId (**args*, ***kwargs*)

GetId(*self*) -> int

GetInterval (**args*, ***kwargs*)

GetInterval(*self*) -> int

GetNextHandler (**args*, ***kwargs*)

GetNextHandler(*self*) -> EvtHandler

GetOwner (**args*, ***kwargs*)

GetOwner(*self*) -> EvtHandler

GetPreviousHandler (**args*, ***kwargs*)

GetPreviousHandler(*self*) -> EvtHandler

Id

See *GetId*

Interval

See *GetInterval*

IsOneShot (*args, **kwargs)

IsOneShot(self) -> bool

IsRunning (*args, **kwargs)

IsRunning(self) -> bool

IsSameAs (*args, **kwargs)

IsSameAs(self, Object p) -> bool

For wx.Objects that use C++ reference counting internally, this method can be used to determine if two objects are referencing the same data object.

IsUnlinked (*args, **kwargs)

IsUnlinked(self) -> bool

NextHandler

See *GetNextHandler* and *SetNextHandler*

Notify ()

Owner

See *GetOwner* and *SetOwner*

PreviousHandler

See *GetPreviousHandler* and *SetPreviousHandler*

ProcessEvent (*args, **kwargs)

ProcessEvent(self, Event event) -> bool

ProcessEventLocally (*args, **kwargs)

ProcessEventLocally(self, Event event) -> bool

ProcessPendingEvents (self)

QueueEvent (self, Event event)

SafelyProcessEvent (*args, **kwargs)

SafelyProcessEvent(self, Event event) -> bool

SetEvtHandlerEnabled (self, bool enabled)

SetNextHandler (self, EvtHandler handler)

SetOwner (self, EvtHandler owner, int id=ID_ANY)

SetPreviousHandler (self, EvtHandler handler)

Start (*args, **kwargs)

Start(self, int milliseconds=-1, bool oneShot=False) -> bool

Stop (self)

Unbind (event, source=None, id=-1, id2=-1, handler=None)

Disconnects the event handler binding for event from self. Returns True if successful.

Unlink (self)

thisown

The membership flag

8.82.3 Functions

`IPython.lib.inputhookwx.inputhook_wx1()`

Run the wx event loop by processing pending events only.

This approach seems to work, but its performance is not great as it relies on having `PyOS_InputHook` called regularly.

`IPython.lib.inputhookwx.inputhook_wx2()`

Run the wx event loop, polling for stdin.

This version runs the wx eventloop for an undetermined amount of time, during which it periodically checks to see if anything is ready on stdin. If anything is ready on stdin, the event loop exits.

The argument to `elr.Run` controls how often the event loop looks at stdin. This determines the responsiveness at the keyboard. A setting of 1000 enables a user to type at most 1 char per second. I have found that a setting of 10 gives good keyboard response. We can shorten it further, but eventually performance would suffer from calling `select/kbhit` too often.

`IPython.lib.inputhookwx.inputhook_wx3()`

Run the wx event loop by processing pending events only.

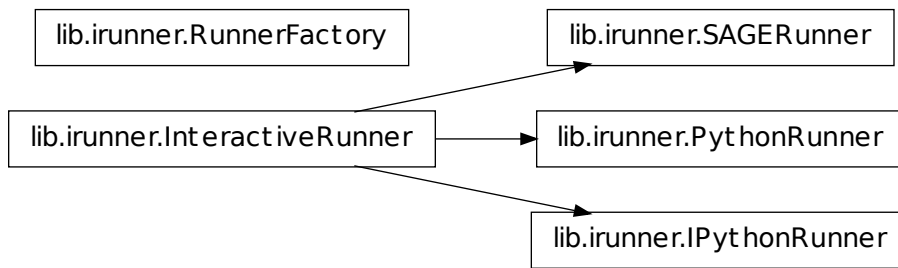
This is like `inputhook_wx1`, but it keeps processing pending events until stdin is ready. After processing all pending events, a call to `time.sleep` is inserted. This is needed, otherwise, CPU usage is at 100%. This sleep time should be tuned though for best performance.

`IPython.lib.inputhookwx.stdin_ready()`

8.83 lib.irunner

8.83.1 Module: `lib.irunner`

Inheritance diagram for `IPython.lib.irunner`:



Module for interactively running scripts.

This module implements classes for interactively running scripts written for any system with a prompt which can be matched by a regexp suitable for pexpect. It can be used to run as if they had been typed up interactively, an arbitrary series of commands for the target system.

The module includes classes ready for IPython (with the default prompts), plain Python and SAGE, but making a new one is trivial. To see how to use it, simply run the module as a script:

```
./irunner.py -help
```

This is an extension of Ken Schutte <kschutte-AT-csail.mit.edu>'s script contributed on the ipython-user list:

<http://scipy.net/pipermail/ipython-user/2006-May/001705.html>

NOTES:

- This module requires pexpect, available in most linux distros, or which can be downloaded from <http://pexpect.sourceforge.net>
- Because pexpect only works under Unix or Windows-Cygwin, this has the same limitations. This means that it will NOT work under native windows Python.

8.83.2 Classes

IPythonRunner

```
class IPython.lib.irunner.IPythonRunner (program='ipython',          args=None,
                                         out=<open file '<stdout>', mode 'w'
                                         at 0x1004160b8>, echo=True)
```

Bases: `IPython.lib.irunner.InteractiveRunner`

Interactive IPython runner.

This initializes IPython in ‘nocolor’ mode for simplicity. This lets us avoid having to write a regexp that matches ANSI sequences, though pexpect does support them. If anyone contributes patches for ANSI color support, they will be welcome.

It also sets the prompts manually, since the prompt regexps for pexpect need to be matched to the actual prompts, so user-customized prompts would break this.

```
__init__(program='ipython', args=None, out=<open file '<stdout>', mode 'w' at
        0x1004160b8>, echo=True)
    New runner, optionally passing the ipython command to use.
```

```
close()
    close child process
```

```
main(argv=None)
    Run as a command-line script.
```

```
run_file(fname, interact=False, get_output=False)
    Run the given file interactively.
```

Inputs:

-fname: name of the file to execute.

See the run_source docstring for the meaning of the optional arguments.

```
run_source(source, interact=False, get_output=False)
    Run the given source code interactively.
```

Inputs:

- source: a string of code to be executed, or an open file object we can iterate over.

Optional inputs:

- interact(False): if true, start to interact with the running program at the end of the script. Otherwise, just exit.
- get_output(False): if true, capture the output of the child process (filtering the input commands out) and return it as a string.

Returns: A string containing the process output, but only if requested.

InteractiveRunner

```
class IPython.lib.irunner.InteractiveRunner(program, prompts, args=None,
        out=<open file '<stdout>', mode 'w'
        at 0x1004160b8>, echo=True)
```

Bases: object

Class to run a sequence of commands through an interactive program.

__init__ (*program, prompts, args=None, out=<open file '<stdout>', mode 'w' at 0x1004160b8>, echo=True*)
Construct a runner.

Inputs:

- **program**: command to execute the given program.
- **prompts**: a list of patterns to match as valid prompts, in the

format used by pexpect. This basically means that it can be either a string (to be compiled as a regular expression) or a list of such (it must be a true list, as pexpect does type checks).

If more than one prompt is given, the first is treated as the main program prompt and the others as ‘continuation’ prompts, like python’s. This means that blank lines in the input source are omitted when the first prompt is matched, but are NOT omitted when the continuation one matches, since this is how python signals the end of multiline input interactively.

Optional inputs:

- **args(None)**: optional list of strings to pass as arguments to the child program.
- **out(sys.stdout)**: if given, an output stream to be used when writing output. The only requirement is that it must have a `.write()` method.

Public members not parameterized in the constructor:

- **delaybeforesend(0)**: Newer versions of pexpect have a delay before sending each new input. For our purposes here, it’s typically best to just set this to zero, but if you encounter reliability problems or want an interactive run to pause briefly at each prompt, just increase this value (it is measured in seconds). Note that this variable is not honored at all by older versions of pexpect.

close()
close child process

main (*argv=None*)
Run as a command-line script.

run_file (*fname, interact=False, get_output=False*)
Run the given file interactively.

Inputs:

-**fname**: name of the file to execute.

See the `run_source` docstring for the meaning of the optional arguments.

run_source (*source, interact=False, get_output=False*)
Run the given source code interactively.

Inputs:

- source: a string of code to be executed, or an open file object we can iterate over.

Optional inputs:

- interact(False): if true, start to interact with the running program at the end of the script. Otherwise, just exit.
- get_output(False): if true, capture the output of the child process (filtering the input commands out) and return it as a string.

Returns: A string containing the process output, but only if requested.

PythonRunner

```
class IPython.lib.irunner.PythonRunner (program='python', args=None, out=<open
                                         file '<stdout>', mode 'w' at 0x1004160b8>,
                                         echo=True)
```

Bases: `IPython.lib.irunner.InteractiveRunner`

Interactive Python runner.

```
__init__ (program='python', args=None, out=<open file '<stdout>', mode 'w' at
          0x1004160b8>, echo=True)
```

New runner, optionally passing the python command to use.

```
close ()
```

close child process

```
main (argv=None)
```

Run as a command-line script.

```
run_file (fname, interact=False, get_output=False)
```

Run the given file interactively.

Inputs:

- fname: name of the file to execute.

See the run_source docstring for the meaning of the optional arguments.

```
run_source (source, interact=False, get_output=False)
```

Run the given source code interactively.

Inputs:

- source: a string of code to be executed, or an open file object we can iterate over.

Optional inputs:

- interact(False): if true, start to interact with the running program at the end of the script. Otherwise, just exit.

•`get_output(False)`: if true, capture the output of the child process (filtering the input commands out) and return it as a string.

Returns: A string containing the process output, but only if requested.

RunnerFactory

```
class IPython.lib.irunner.RunnerFactory (out=<open file '<stdout>', mode 'w' at
                                         0x1004160b8>)
```

Bases: object

Code runner factory.

This class provides an IPython code runner, but enforces that only one runner is ever instantiated. The runner is created based on the extension of the first file to run, and it raises an exception if a runner is later requested for a different extension type.

This ensures that we don't generate example files for doctest with a mix of python and ipython syntax.

```
__init__ (out=<open file '<stdout>', mode 'w' at 0x1004160b8>)
    Instantiate a code runner.
```

SAGERunner

```
class IPython.lib.irunner.SAGERunner (program='sage', args=None, out=<open
                                         file '<stdout>', mode 'w' at 0x1004160b8>,
                                         echo=True)
```

Bases: `IPython.lib.irunner.InteractiveRunner`

Interactive SAGE runner.

WARNING: this runner only works if you manually configure your SAGE copy to use 'colors No-Color' in the ipythonrc config file, since currently the prompt matching regexp does not identify color sequences.

```
__init__ (program='sage', args=None, out=<open file '<stdout>', mode 'w' at
                                         0x1004160b8>, echo=True)
    New runner, optionally passing the sage command to use.
```

```
close ()
    close child process
```

```
main (argv=None)
    Run as a command-line script.
```

```
run_file (fname, interact=False, get_output=False)
    Run the given file interactively.
```

Inputs:

-fname: name of the file to execute.

See the `run_source` docstring for the meaning of the optional arguments.

run_source (*source*, *interact=False*, *get_output=False*)

Run the given source code interactively.

Inputs:

- *source*: a string of code to be executed, or an open file object we can iterate over.

Optional inputs:

- *interact(False)*: if true, start to interact with the running program at the end of the script. Otherwise, just exit.
- *get_output(False)*: if true, capture the output of the child process (filtering the input commands out) and return it as a string.

Returns: A string containing the process output, but only if requested.

8.83.3 Functions

`IPython.lib.irunner.main()`

Run as a command-line script.

`IPython.lib.irunner.pexpect_monkeypatch()`

Patch pexpect to prevent unhandled exceptions at VM teardown.

Calling this function will monkeypatch the `pexpect.spawn` class and modify its `__del__` method to make it more robust in the face of failures that can occur if it is called when the Python VM is shutting down.

Since Python may fire `__del__` methods arbitrarily late, it's possible for them to execute during the teardown of the Python VM itself. At this point, various builtin modules have been reset to `None`. Thus, the call to `self.close()` will trigger an exception because it tries to call `os.close()`, and `os` is now `None`.

8.84 lib.latextools

8.84.1 Module: `lib.latextools`

Tools for handling LaTeX.

Authors:

- Brian Granger

8.84.2 Functions

`IPython.lib.latextools.latex_to_html(s, alt='image')`

Render LaTeX to HTML with embedded PNG data using data URIs.

Parameters `s` : str

The raw string containing valid inline LaTeX.

`alt` : str

The alt text to use for the HTML.

`IPython.lib.latextools.latex_to_png(s, encode=True)`

Render a LaTeX string to PNG using matplotlib.mathtext.

Parameters `s` : str

The raw string containing valid inline LaTeX.

`encode` : bool, optional

Should the PNG data bebase64 encoded to make it JSON'able.

8.85 lib.pylabtools

8.85.1 Module: `lib.pylabtools`

Pylab (matplotlib) support utilities.

Authors

- Fernando Perez.
- Brian Granger

8.85.2 Functions

`IPython.lib.pylabtools.activate_matplotlib(backend)`

Activate the given backend and set interactive to True.

`IPython.lib.pylabtools.figure_size(sizex, sizey)`

Set the default figure size to be [sizex, sizey].

This is just an easy to remember, convenience wrapper that sets:

```
matplotlib.rcParams['figure.figsize'] = [sizex, sizey]
```

`IPython.lib.pylabtools.figure_to_svg(fig)`

Convert a figure to svg for inline display.

`IPython.lib.pylabtools.find_gui_and_backend(gui=None)`

Given a gui string return the gui and mpl backend.

Parameters `gui` : str

Can be one of ('tk','gtk','wx','qt','qt4','inline').

Returns A tuple of (gui, backend) where backend is one of ('TkAgg','GTKAgg', :
'WXAgg','Qt4Agg','module://IPython.zmq.pylab.backend_inline'). :

`IPython.lib.pylabtools.getfigs(*fig_nums)`

Get a list of matplotlib figures by figure numbers.

If no arguments are given, all available figures are returned. If the argument list contains references to invalid figures, a warning is printed but the function continues pasting further figures.

Parameters `figs` : tuple

A tuple of ints giving the figure numbers of the figures to return.

`IPython.lib.pylabtools.import_pylab(user_ns, backend, import_all=True, shell=None)`

Import the standard pylab symbols into user_ns.

`IPython.lib.pylabtools.mpl_runner(safe_execfile)`

Factory to return a matplotlib-enabled runner for %run.

Parameters `safe_execfile` : function

This must be a function with the same interface as the `safe_execfile()` method of IPython.

Returns A function suitable for use as the “runner“ argument of the %run magic :
function. :

`IPython.lib.pylabtools.pylab_activate(user_ns, gui=None, import_all=True)`

Activate pylab mode in the user's namespace.

Loads and initializes numpy, matplotlib and friends for interactive use.

Parameters `user_ns` : dict

Namespace where the imports will occur.

gui : optional, string

A valid gui name following the conventions of the %gui magic.

import_all : optional, boolean

If true, an 'import *' is done from numpy and pylab.

Returns The actual gui used (if not given as input, it was obtained from matplotlib :
itself, and will be needed next to configure IPython's gui integration. :

8.86 testing

8.86.1 Module: `testing`

Testing support (tools to test IPython itself).

`IPython.testing.test()`

Run the entire IPython test suite.

For fine-grained control, you should use the `iptest` script supplied with the IPython installation.

8.87 testing.decorators

8.87.1 Module: `testing.decorators`

Decorators for labeling test objects.

Decorators that merely return a modified version of the original function object are straightforward. Decorators that return a new function object need to use `nose.tools.make_decorator(original_function)(decorator)` in returning the decorator, in order to preserve metadata such as function name, setup and teardown functions and so on - see `nose.tools` for more information.

This module provides a set of useful decorators meant to be ready to use in your own tests. See the bottom of the file for the ready-made ones, and if you find yourself writing a new one that may be of generic use, add it here.

Included decorators:

Lightweight testing that remains unittest-compatible.

- `@parametric`, for parametric test support that is vastly easier to use than nose's for debugging. With ours, if a test fails, the stack under inspection is that of the test and not that of the test framework.
- An `@as_unittest` decorator can be used to tag any normal parameter-less function as a unittest Test-Case. Then, both nose and normal unittest will recognize it as such. This will make it easier to migrate away from Nose if we ever need/want to while maintaining very lightweight tests.

NOTE: This file contains IPython-specific decorators and imports the `numpy.testing.decorators` file, which we've copied verbatim. Any of our own code will be added at the bottom if we end up extending this.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.87.2 Functions

`IPython.testing.decorators.apply_wrapper(wrapper, func)`

Apply a wrapper to a function for decoration.

This mixes Michele Simionato's decorator tool with nose's `make_decorator`, to apply a wrapper in a decorator so that all nose attributes, as well as function signature and other properties, survive the decoration cleanly. This will ensure that wrapped functions can still be well introspected via IPython, for example.

`IPython.testing.decorators.as_unittest` (*func*)

Decorator to make a simple function into a normal test via unittest.

`IPython.testing.decorators.make_label_dec` (*label, ds=None*)

Factory function to create a decorator that applies one or more labels.

Parameters *label* : string or sequence

One or more labels that will be applied by the decorator to the functions

it decorates. Labels are attributes of the decorated function with their :

value set to True. :

ds : string An optional docstring for the resulting decorator. If not given, a default docstring is auto-generated.

Returns A decorator. :

Examples

A simple labeling decorator: `>>> slow = make_label_dec('slow') >>> print slow.__doc__` Labels a test as 'slow'.

And one that uses multiple labels and a custom docstring: `>>> rare = make_label_dec(['slow', 'hard'], ... "Mix labels 'slow' and 'hard' for rare tests.") >>> print rare.__doc__` Mix labels 'slow' and 'hard' for rare tests.

Now, let's test using this one: `>>> @rare ... def f(): pass ... >>> >>> f.slow True >>> f.hard True`

`IPython.testing.decorators.numpy_not_available` ()

Can numpy be imported? Returns true if numpy does NOT import.

This is used to make a decorator to skip tests that require numpy to be available, but delay the 'import numpy' to test execution time.

`IPython.testing.decorators.onlyif` (*condition, msg*)

The reverse from skipif, see skipif for details.

`IPython.testing.decorators.skip` (*msg=None*)

Decorator factory - mark a test function for skipping from test suite.

Parameters *msg* : string

Optional message to be added.

Returns *decorator* : function

Decorator, which, when applied to a function, causes `SkipTest` to be raised, with the optional message added.

`IPython.testing.decorators.skipif` (*skip_condition*, *msg=None*)

Make function raise `SkipTest` exception if *skip_condition* is true

Parameters *skip_condition* : bool or callable.

Flag to determine whether to skip test. If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed. *msg* : string

Message to give on raising a `SkipTest` exception

Returns *decorator* : function

Decorator, which, when applied to a function, causes `SkipTest` to be raised when the *skip_condition* was `True`, and the function to be called normally otherwise.

Notes

You will see from the code that we had to further decorate the decorator with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

8.88 testing.decorators_trial

8.88.1 Module: `testing.decorators_trial`

Testing related decorators for use with `twisted.trial`.

The decorators in this files are designed to follow the same API as those in the `decorators` module (in this same directory). But they can be used with `twisted.trial`

8.88.2 Functions

`IPython.testing.decorators_trial.numpy_not_available()`

Can numpy be imported? Returns true if numpy does NOT import.

This is used to make a decorator to skip tests that require numpy to be available, but delay the ‘import numpy’ to test execution time.

`IPython.testing.decorators_trial.skip` (*msg=None*)

Create a decorator that marks a test function for skipping.

This is a decorator factory that returns a decorator that will cause tests to be skipped.

Parameters *msg* : str

Optional message to be added.

Returns *decorator* : function

Decorator, which, when applied to a function, sets the skip attribute of the function causing *twisted.trial* to skip it.

`IPython.testing.decorators_trial.skipif(skip_condition, msg=None)`

Create a decorator that marks a test function for skipping.

There is a decorator factory that returns a decorator that will conditionally skip a test based on the value of `skip_condition`. The `skip_condition` argument can either be a boolean or a callable that returns a boolean.

Parameters `skip_condition` : boolean or callable

If this evaluates to True, the test is skipped.

`msg` : str

The message to print if the test is skipped.

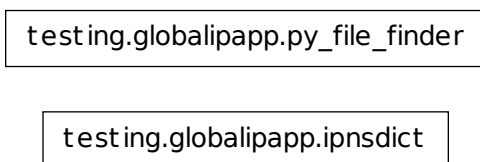
Returns `decorator` : function

The decorator function that can be applied to the test function.

8.89 testing.globalipapp

8.89.1 Module: `testing.globalipapp`

Inheritance diagram for `IPython.testing.globalipapp`:



Global IPython app to support test running.

We must start our own `ipython` object and heavily muck with it so that all the modifications IPython makes to system behavior don't send the doctest machinery into a fit. This code should be considered a gross hack, but it gets the job done.

8.89.2 Classes

`ipnsdict`

class `IPython.testing.globalipapp.ipnsdict (*a)`

Bases: `dict`

A special subclass of `dict` for use as an IPython namespace in doctests.

This subclass adds a simple checkpointing capability so that when testing machinery clears it (we use it as the test execution context), it doesn't get completely destroyed.

In addition, it can handle the presence of the `'_'` key in a special manner, which is needed because of how Python's doctest machinery operates with `'_'`. See constructor and `update()` for details.

`__init__` (`*a`)

`clear` ()

`copy`

`D.copy()` -> a shallow copy of `D`

`static fromkeys` ()

`dict.fromkeys(S,v)` -> New dict with keys from `S` and values equal to `v`. `v` defaults to `None`.

`get`

`D.get(k,d)` -> `D[k]` if `k` in `D`, else `d`. `d` defaults to `None`.

`has_key`

`D.has_key(k)` -> True if `D` has a key `k`, else False

`items`

`D.items()` -> list of `D`'s (key, value) pairs, as 2-tuples

`iteritems`

`D.iteritems()` -> an iterator over the (key, value) items of `D`

`iterkeys`

`D.iterkeys()` -> an iterator over the keys of `D`

`intervalues`

`D.intervalues()` -> an iterator over the values of `D`

`keys`

`D.keys()` -> list of `D`'s keys

`pop`

`D.pop(k,d)` -> `v`, remove specified key and return the corresponding value. If key is not found, `d` is returned if given, otherwise `KeyError` is raised

`popitem`

`D.popitem()` -> (`k`, `v`), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if `D` is empty.

`setdefault`

`D.setdefault(k,d)` -> `D.get(k,d)`, also set `D[k]=d` if `k` not in `D`

update (*other*)

values

D.values() -> list of D's values

py_file_finder

class IPython.testing.globalipapp.**py_file_finder** (*test_filename*)

Bases: object

__init__ (*test_filename*)

8.89.3 Functions

IPython.testing.globalipapp.**get_ipython** ()

IPython.testing.globalipapp.**start_ipython** ()

Start a global IPython shell, which we need for IPython-specific syntax.

IPython.testing.globalipapp.**xsys** (*self*, *cmd*)

Replace the default system call with a capturing one for doctest.

8.90 testing.iptest

8.90.1 Module: testing.iptest

Inheritance diagram for IPython.testing.iptest:

testing.iptest.IPTester

IPython Test Suite Runner.

This module provides a main entry point to a user script to test IPython itself from the command line. There are two ways of running this script:

1. With the syntax *iptest all*. This runs our entire test suite by calling this script (with different arguments) or trial recursively. This causes modules and package to be tested in different processes, using nose or trial where appropriate.
2. With the regular nose syntax, like *iptest -vvs IPython*. In this form the script simply calls nose, but with special command line flags and plugins loaded.

For now, this script requires that both nose and twisted are installed. This will change in the future.

8.90.2 Class

8.90.3 IPTester

class IPython.testing.iptest.**IPTester** (*runner='iptest', params=None*)

Bases: object

Call that calls iptest or trial in a subprocess.

__init__ (*runner='iptest', params=None*)

Create new test runner.

call_args

list, arguments of system call to be made to call test runner

params

list, parameters for test runner

pids

list, process ids of subprocesses we start (for cleanup)

run ()

Run the stored commands

runner

string, name of test runner that will be called

8.90.4 Functions

IPython.testing.iptest.**main** ()

IPython.testing.iptest.**make_exclude** ()

Make patterns of modules and packages to exclude from testing.

For the IPythonDoctest plugin, we need to exclude certain patterns that cause testing problems. We should strive to minimize the number of skipped modules, since this means untested code.

These modules and packages will NOT get scanned by nose at all for tests.

IPython.testing.iptest.**make_runners** ()

Define the top-level packages that need to be tested.

IPython.testing.iptest.**report** ()

Return a string with a summary report of test-related variables.

IPython.testing.iptest.**run_iptest** ()

Run the IPython test suite using nose.

This function is called when this script is **not** called with the form *iptest all*. It simply calls nose with appropriate command line flags and accepts all of the standard nose arguments.

IPython.testing.iptest.**run_iptestall** ()

Run the entire IPython test suite by calling nose and trial.

This function constructs `IPTester` instances for all IPython modules and package and then runs each of them. This causes the modules and packages of IPython to be tested each in their own sub-process using nose or twisted.trial appropriately.

`IPython.testing.iptest.test_for(mod)`
Test to see if mod is importable.

8.91 testing.ipunittest

8.91.1 Module: testing.ipunittest

Inheritance diagram for `IPython.testing.ipunittest`:

`testing.ipunittest.IPython2PythonConverter`

`testing.ipunittest.Doc2UnitTester`

Experimental code for cleaner support of IPython syntax with unittest.

In IPython up until 0.10, we've used very hacked up nose machinery for running tests with IPython special syntax, and this has proved to be extremely slow. This module provides decorators to try a different approach, stemming from a conversation Brian and I (FP) had about this problem Sept/09.

The goal is to be able to easily write simple functions that can be seen by unittest as tests, and ultimately for these to support doctests with full IPython syntax. Nose already offers this based on naming conventions and our hackish plugins, but we are seeking to move away from nose dependencies if possible.

This module follows a different approach, based on decorators.

- A decorator called `@ipdoctest` can mark any function as having a docstring that should be viewed as a doctest, but after syntax conversion.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.91.2 Classes

Doc2UnitTester

class IPython.testing.ipunittest.**Doc2UnitTester** (*verbose=False*)

Bases: object

Class whose instances act as a decorator for docstring testing.

In practice we're only likely to need one instance ever, made below (though no attempt is made at turning it into a singleton, there is no need for that).

__init__ (*verbose=False*)

New decorator.

Parameters **verbose** : boolean, optional (False)

Passed to the doctest finder and runner to control verbosity.

IPython2PythonConverter

class IPython.testing.ipunittest.**IPython2PythonConverter**

Bases: object

Convert IPython 'syntax' to valid Python.

Eventually this code may grow to be the full IPython syntax conversion implementation, but for now it only does prompt conversion.

__init__ ()

8.91.3 Functions

IPython.testing.ipunittest.**count_failures** (*runner*)

Count number of failures in a doctest runner.

Code modeled after the summarize() method in doctest.

IPython.testing.ipunittest.**ipdocstring** (*func*)

Change the function docstring via ip2py.

8.92 testing.mkdoctests

8.92.1 Module: testing.mkdoctests

Inheritance diagram for IPython.testing.mkdoctests:

testing.mkdoctests.IndentOut

testing.mkdoctests.RunnerFactory

Utility for making a doctest file out of Python or IPython input.

```
%prog [options] input_file [output_file]
```

This script is a convenient generator of doctest files that uses IPython's `irunner` script to execute valid Python or IPython input in a separate process, capture all of the output, and write it to an output file.

It can be used in one of two ways:

1. With a plain Python or IPython input file (denoted by extensions `.py` or `.ipy`). In this case, the output is an auto-generated reST file with a basic header, and the captured Python input and output contained in an indented code block.

If no output filename is given, the input name is used, with the extension replaced by `.txt`.

2. With an input template file. Template files are simply plain text files with special directives of the form

```
%run filename
```

to include the named file at that point.

If no output filename is given and the input filename is of the form `'base.tpl.txt'`, the output will be automatically named `'base.txt'`.

8.92.2 Classes

IndentOut

class `IPython.testing.mkdoctests.IndentOut` (*out=<open file '<stdout>', mode 'w' at 0x1004160b8>, indent=4*)

Bases: `object`

A simple output stream that indents all output by a fixed amount.

Instances of this class trap output to a given stream and first reformat it to indent every input line.

__init__ (*out=<open file '<stdout>', mode 'w' at 0x1004160b8>, indent=4*)

Create an indented writer.

Keywords

- *out* : stream (`sys.stdout`) Output stream to actually write to after indenting.

•*indent* : int Number of spaces to indent every input line by.

close()

flush()

write(data)

Write a string to the output stream.

RunnerFactory

class IPython.testing.mkdoctests.**RunnerFactory** (*out=<open file '<stdout>', mode 'w' at 0x1004160b8>*)

Bases: object

Code runner factory.

This class provides an IPython code runner, but enforces that only one runner is every instantiated. The runner is created based on the extension of the first file to run, and it raises an exception if a runner is later requested for a different extension type.

This ensures that we don't generate example files for doctest with a mix of python and ipython syntax.

__init__ (*out=<open file '<stdout>', mode 'w' at 0x1004160b8>*)

Instantiate a code runner.

8.92.3 Function

IPython.testing.mkdoctests.**main()**

Run as a script.

8.93 testing.nosepatch

8.93.1 Module: testing.nosepatch

Monkeypatch nose to accept any callable as a method.

By default, nose's ismethod() fails for static methods. Once this is fixed in upstream nose we can disable it.

Note: merely importing this module causes the monkeypatch to be applied.

IPython.testing.nosepatch.**getTestCaseNames** (*self, testCaseClass*)

Override to select with selector, unless config.getTestCaseNamesCompat is True

8.94 testing.parametric

8.94.1 Module: `testing.parametric`

Parametric testing on top of `twisted.trial.unittest`.

XXX - It may be possible to deprecate this in favor of the new, cleaner parametric code. We just need to double-check that the new code doesn't clash with Twisted (we know it works with nose and unittest).

8.94.2 Functions

`IPython.testing.parametric.Parametric(cls)`

Register parametric tests with a class.

`IPython.testing.parametric.parametric(f)`

Mark `f` as a parametric test.

`IPython.testing.parametric.partial(f, *partial_args, **partial_kwargs)`

Generate a partial class method.

8.95 testing.plugin.dtexample

8.95.1 Module: `testing.plugin.dtexample`

Simple example using doctests.

This file just contains doctests both using plain python and IPython prompts. All tests should be loaded by nose.

8.95.2 Functions

`IPython.testing.plugin.dtexample.ipfunc()`

Some ipython tests...

In [1]: `import os`

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

In [26]: `for i in range(3):: print i,: print i+1,:`

`0 1 1 2 2 3`

Examples that access the operating system work:

In [1]: `!echo hello hello`

In [2]: `!echo hello > /tmp/foo`

In [3]: `!cat /tmp/foo hello`

In [4]: `rm -f /tmp/foo`

It's OK to use '_' for the last result, but do NOT try to use IPython's numbered history of _NN outputs, since those won't exist under the doctest environment:

```
In [7]: 'hi' Out[7]: 'hi'
```

```
In [8]: print repr(_) 'hi'
```

```
In [7]: 3+4 Out[7]: 7
```

```
In [8]: _+3 Out[8]: 10
```

```
In [9]: ipfunc() Out[9]: 'ipfunc'
```

```
IPython.testing.plugin.doctestexample.iprand()
```

Some ipython tests with random output.

```
In [7]: 3+4 Out[7]: 7
```

```
In [8]: print 'hello' world # random
```

```
In [9]: iprand() Out[9]: 'iprand'
```

```
IPython.testing.plugin.doctestexample.iprand_all()
```

Some ipython tests with fully random output.

```
# all-random
```

```
In [7]: 1 Out[7]: 99
```

```
In [8]: print 'hello' world
```

```
In [9]: iprand_all() Out[9]: 'junk'
```

```
IPython.testing.plugin.doctestexample.pyfunc()
```

Some pure python tests...

```
>>> pyfunc()
'pyfunc'
```

```
>>> import os
```

```
>>> 2+3
5
```

```
>>> for i in range(3):
...     print i,
...     print i+1,
...
0 1 1 2 2 3
```

```
IPython.testing.plugin.doctestexample.random_all()
```

A function where we ignore the output of ALL examples.

Examples:

```
# all-random
```

This mark tells the testing machinery that all subsequent examples should be treated as random (ignoring their output). They are still executed, so if a they raise an error, it will be detected as such, but their output is completely ignored.

```
>>> 1+3
junk goes here...

>>> 1+3
klasdfj;

>>> 1+2
again, anything goes
blah...
```

`IPython.testing.plugin.dtexample.ranfunc()`

A function with some random output.

Normal examples are verified as usual: `>>> 1+3 4`

But if you put ‘# random’ in the output, it is ignored: `>>> 1+3 junk goes here... # random`

```
>>> 1+2
again, anything goes #random
if multiline, the random mark is only needed once.
```

```
>>> 1+2
You can also put the random marker at the end:
# random
```

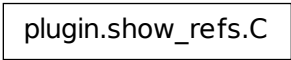
```
>>> 1+2
# random
.. or at the beginning.
```

More correct input is properly verified: `>>> ranfunc() ‘ranfunc’`

8.96 testing.plugin.show_refs

8.96.1 Module: testing.plugin.show_refs

Inheritance diagram for `IPython.testing.plugin.show_refs`:



```
graph TD
    A[plugin.show_refs.C]
```

Simple script to show reference holding behavior.

This is used by a companion test case.

8.96.2 c

class IPython.testing.plugin.show_refs.C

Bases: object

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

8.97 testing.plugin.simple

8.97.1 Module: testing.plugin.simple

Simple example using doctests.

This file just contains doctests both using plain python and IPython prompts. All tests should be loaded by nose.

8.97.2 Functions

IPython.testing.plugin.simple.**ipyfunc2**()

Some pure python tests...

```
>>> 1+1
2
```

IPython.testing.plugin.simple.**pyfunc**()

Some pure python tests...

```
>>> pyfunc()
'pyfunc'
```

```
>>> import os
```

```
>>> 2+3
5
```

```
>>> for i in range(3):
...     print i,
...     print i+1,
...
0 1 1 2 2 3
```

8.98 testing.plugin.test_ipdoctest

8.98.1 Module: testing.plugin.test_ipdoctest

Tests for the ipdoctest machinery itself.

Note: in a file named test_X, functions whose only test is their docstring (as a doctest) and which have no test functionality of their own, should be called 'doctest_foo' instead of 'test_foo', otherwise they get double-counted (the empty function call is counted as a test, which just inflates tests numbers artificially).

8.98.2 Functions

IPython.testing.plugin.test_ipdoctest.**doctest_multiline1**()

The ipdoctest machinery must handle multiline examples gracefully.

In [2]: for i in range(10): ...: print i, ...:

0 1 2 3 4 5 6 7 8 9

IPython.testing.plugin.test_ipdoctest.**doctest_multiline2**()

Multiline examples that define functions and print output.

In [7]: def f(x): ...: return x+1 ...:

In [8]: f(1) Out[8]: 2

In [9]: def g(x): ...: print 'x is:',x ...:

In [10]: g(1) x is: 1

In [11]: g('hello') x is: hello

IPython.testing.plugin.test_ipdoctest.**doctest_multiline3**()

Multiline examples with blank lines.

In [12]: def h(x):: if x>1:: return x**2: # To leave a blank line in the input, you must mark
it: # with a comment character:: #: # otherwise the doctest parser gets confused.:
else:: return -1:

In [13]: h(5) Out[13]: 25

In [14]: h(1) Out[14]: -1

In [15]: h(0) Out[15]: -1

IPython.testing.plugin.test_ipdoctest.**doctest_simple**()

ipdoctest must handle simple inputs

In [1]: 1 Out[1]: 1

In [2]: print 1 1

8.99 testing.plugin.test_refs

8.99.1 Module: testing.plugin.test_refs

Some simple tests for the plugin while running scripts.

8.99.2 Functions

`IPython.testing.plugin.test_refs.doctest_ivars()`
 Test that variables defined interactively are picked up. In [5]: zz=1

In [6]: zz Out[6]: 1

`IPython.testing.plugin.test_refs.doctest_refs()`
 DocTest reference holding issues when running scripts.

In [32]: run show_refs.py c referrers: [<type 'dict'>]

`IPython.testing.plugin.test_refs.doctest_run()`
 Test running a trivial script.

In [13]: run simplevars.py x is: 1

`IPython.testing.plugin.test_refs.doctest_runvars()`
 Test that variables defined in scripts get loaded correctly via `%run`.

In [13]: run simplevars.py x is: 1

In [14]: x Out[14]: 1

`IPython.testing.plugin.test_refs.test_trivial()`
 A trivial passing test.

8.100 testing.tools

8.100.1 Module: `testing.tools`

Inheritance diagram for `IPython.testing.tools`:

`testing.tools.TempFileMixin`

Generic testing tools that do NOT depend on Twisted.

In particular, this module exposes a set of top-level `assert*` functions that can be used in place of `nose.tools.assert*` in method generators (the ones in `nose` can not, at least as of `nose 0.10.4`).

Note: our testing package contains `testing.util`, which does depend on Twisted and provides utilities for tests that manage `Deferreds`. All testing support tools that only depend on `nose`, `IPython` or the standard library should go here instead.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.100.2 Class

8.100.3 TempFileMixin

class IPython.testing.tools.TempFileMixin

Bases: object

Utility class to create temporary Python/IPython files.

Meant as a mixin class for test cases.

__init__ ()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

mktmp (src, ext='.py')

Make a valid python temp file.

tearDown ()

8.100.4 Functions

IPython.testing.tools.default_argv ()

Return a valid default argv for creating testing instances of ipython

IPython.testing.tools.default_config ()

Return a config object with good defaults for testing.

IPython.testing.tools.full_path (startPath, files)

Make full paths for all the listed files, based on startPath.

Only the base part of startPath is kept, since this routine is typically used with a script's `__file__` variable as startPath. The base of startPath is then prepended to all the listed files, forming the output list.

Parameters startPath : string

Initial path to use as the base for the results. This path is split

using `os.path.split()` and only its first component is kept.

files [string or list] One or more files.

Examples

```
>>> full_path('/foo/bar.py', ['a.txt', 'b.txt'])
['/foo/a.txt', '/foo/b.txt']
```

```
>>> full_path('/foo', ['a.txt', 'b.txt'])
['/a.txt', '/b.txt']
```

If a single file is given, the output is still a list: >>> full_path('/foo', 'a.txt') ['/a.txt']

IPython.testing.tools.**ipexec** (*fname*, *options=None*)

Utility to call 'ipython filename'.

Starts IPython with a minimal and safe configuration to make startup as fast as possible.

Note that this starts IPython in a subprocess!

Parameters **fname** : str

Name of file to be executed (should have .py or .ipy extension).

options : optional, list

Extra command-line flags to be passed to IPython.

Returns (stdout, stderr) of ipython subprocess. :

IPython.testing.tools.**ipexec_validate** (*fname*, *expected_out*, *expected_err=''*, *options=None*)

Utility to call 'ipython filename' and validate output/error.

This function raises an AssertionError if the validation fails.

Note that this starts IPython in a subprocess!

Parameters **fname** : str

Name of the file to be executed (should have .py or .ipy extension).

expected_out : str

Expected stdout of the process.

expected_err : optional, str

Expected stderr of the process.

options : optional, list

Extra command-line flags to be passed to IPython.

Returns None :

IPython.testing.tools.**parse_test_output** (*txt*)

Parse the output of a test run and return errors, failures.

Parameters **txt** : str

Text output of a test run, assumed to contain a line of one of the following forms:

```
'FAILED (errors=1)'
'FAILED (failures=1)'
'FAILED (errors=1, failures=1)'
```

Returns `nerr`, `nfail`: number of errors and failures. :

8.101 `utils.PyColorize`

8.101.1 Module: `utils.PyColorize`

Inheritance diagram for `IPython.utils.PyColorize`:



```
graph TD; A[utils.PyColorize.Parser];
```

Class and program to colorize python source code for ANSI terminals.

Based on an HTML code highlighter by Jurgen Hermann found at:
<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52298>

Modifications by Fernando Perez (fperez@colorado.edu).

Information on the original HTML highlighter follows:

MoinMoin - Python Source Parser

Title: Colorize Python source using the built-in tokenizer

Submitter: Jurgen Hermann Last Updated:2001/04/06

Version no:1.2

Description:

This code is part of MoinMoin (<http://moin.sourceforge.net/>) and converts Python source code to HTML markup, rendering comments, keywords, operators, numeric and string literals in different colors.

It shows how to use the built-in keyword, token and tokenize modules to scan Python source code and re-emit it with no changes to its original formatting (which is the hard part).

8.101.2 Parser

```
class IPython.utils.PyColorize.Parser (color_table=None, out=<open file '<stdout>',  
                                         mode 'w' at 0x1004160b8>)
```

Format colored Python source.

```
__init__ (color_table=None, out=<open file '<stdout>', mode 'w' at 0x1004160b8>)
```

Create a parser with a specified color table and output channel.

Call `format()` to process code.

```
format (raw, out=None, scheme='')
```


format2 (*raw*, *out=None*, *scheme=''*)

Parse and send the colored source.

If out and scheme are not specified, the defaults (given to constructor) are used.

out should be a file-type object. Optionally, out can be given as the string 'str' and the parser will automatically return the output in a string.

`IPython.utils.PyColorize.main` (*argv=None*)

Run as a command-line script: colorize a python file or stdin using ANSI color escapes and print to stdout.

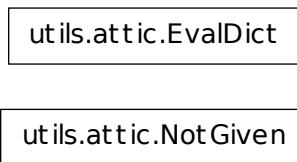
Inputs:

- *argv*(None): a list of strings like `sys.argv[1:]` giving the command-line arguments. If None, use `sys.argv[1:]`.

8.102 utils.attic

8.102.1 Module: `utils.attic`

Inheritance diagram for `IPython.utils.attic`:



Older utilities that are not being used.

WARNING: IF YOU NEED TO USE ONE OF THESE FUNCTIONS, PLEASE FIRST MOVE IT TO ANOTHER APPROPRIATE MODULE IN `IPython.utils`.

8.102.2 Classes

EvalDict

class `IPython.utils.attic.EvalDict`

Emulate a dict which evaluates its contents in the caller's frame.

Usage: `>>> number = 19`

```
>>> text = "python"

>>> print "%(text.capitalize())s %(number/9.0).1f rules!" % EvalDict()
Python 2.1 rules!
```

NotGiven

```
class IPython.utils.attic.NotGiven
```

8.102.3 Functions

`IPython.utils.attic.all_belong(candidates, checklist)`

Check whether a list of items ALL appear in a given list of options.

Returns a single 1 or 0 value.

`IPython.utils.attic.belong(candidates, checklist)`

Check whether a list of items appear in a given list of options.

Returns a list of 1 and 0, one for each candidate given.

`IPython.utils.attic.import_fail_info(mod_name, fns=None)`

Inform load failure for a module.

`IPython.utils.attic.map_method(method, object_list, *argseq, **kw)`

`map_method(method, object_list, *args, **kw) -> list`

Return a list of the results of applying the methods to the items of the argument sequence(s). If more than one sequence is given, the method is called with an argument list consisting of the corresponding item of each sequence. All sequences must be of the same length.

Keyword arguments are passed verbatim to all objects called.

This is Python code, so it's not nearly as fast as the builtin `map()`.

`IPython.utils.attic.mutex_opts(dict, ex_op)`

Check for presence of mutually exclusive keys in a dict.

Call: `mutex_opts(dict, [[op1a, op1b], [op2a, op2b], ...])`

`IPython.utils.attic.popkey(dct, key, default=<class IPython.utils.attic.NotGiven at 0x106d2d830>)`

Return `dct[key]` and delete `dct[key]`.

If default is given, return it if `dct[key]` doesn't exist, otherwise raise `KeyError`.

`IPython.utils.attic.with_obj(object, **args)`

Set multiple attributes for an object, similar to Pascal's `with`.

Example: `with_obj(jim,`

`born = 1960, haircolour = 'Brown', eyecolour = 'Green')`

Credit: Greg Ewing, in <http://mail.python.org/pipermail/python-list/2001-May/040703.html>.

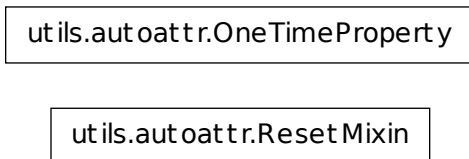
NOTE: up until IPython 0.7.2, this was called simply ‘with’, but ‘with’ has become a keyword for Python 2.5, so we had to rename it.

```
IPython.utils.attic.wrap_deprecated(func, suggest='<nothing>')
```

8.103 utils.autoattr

8.103.1 Module: `utils.autoattr`

Inheritance diagram for `IPython.utils.autoattr`:



Descriptor utilities.

Utilities to support special Python descriptors [1,2], in particular the use of a useful pattern for properties we call ‘one time properties’. These are object attributes which are declared as properties, but become regular attributes once they’ve been read the first time. They can thus be evaluated later in the object’s life cycle, but once evaluated they become normal, static attributes with no function call overhead on access or any other constraints.

A special `ResetMixin` class is provided to add a `.reset()` method to users who may want to have their objects capable of resetting these computed properties to their ‘untriggered’ state.

References

[1] How-To Guide for Descriptors, Raymond Hettinger. <http://users.rcn.com/python/download/Descriptor.htm>

[2] Python data model, <http://docs.python.org/reference/datamodel.html>

Notes

This module is taken from the NiPy project (<http://neuroimaging.scipy.org/site/index.html>), and is BSD licensed.

Authors

- Fernando Perez.

8.103.2 Classes

OneTimeProperty

class IPython.utils.autoattr.**OneTimeProperty** (*func*)

Bases: object

A descriptor to make special properties that become normal attributes.

This is meant to be used mostly by the `auto_attr` decorator in this module.

__init__ (*func*)

Create a OneTimeProperty instance.

Parameters **func** : method

The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

ResetMixin

class IPython.utils.autoattr.**ResetMixin**

Bases: object

A Mixin class to add a `.reset()` method to users of OneTimeProperty.

By default, auto attributes once computed, become static. If they happen to depend on other parts of an object and those parts change, their values may now be invalid.

This class offers a `.reset()` method that users can call *explicitly* when they know the state of their objects may have changed and they want to ensure that *all* their special attributes should be invalidated. Once `reset()` is called, all their auto attributes are reset to their OneTimeProperty descriptors, and their accessor functions will be triggered again.

__init__ ()

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature

reset ()

Reset all OneTimeProperty attributes that may have fired already.

8.103.3 Function

IPython.utils.autoattr.**auto_attr** (*func*)

Decorator to create OneTimeProperty attributes.

Parameters **func** : method

The method that will be called the first time to compute a value. Afterwards, the method's name will be a standard attribute holding the value of this computation.

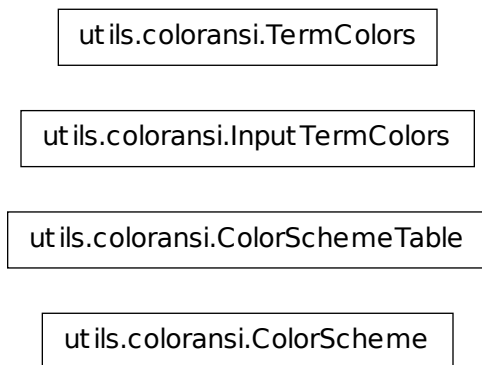
Examples

```
>>> class MagicProp(object):
...     @auto_attr
...     def a(self):
...         return 99
...
>>> x = MagicProp()
>>> 'a' in x.__dict__
False
>>> x.a
99
>>> 'a' in x.__dict__
True
```

8.104 utils.coloransi

8.104.1 Module: `utils.coloransi`

Inheritance diagram for `IPython.utils.coloransi`:



Tools for coloring text in ANSI terminals.

8.104.2 Classes

ColorScheme

```
class IPython.utils.coloransi.ColorScheme (_ColorScheme__scheme_name_, color-  
                                           dict=None, **colormap)  
    Generic color scheme class. Just a name and a Struct.  
  
    __init__ (_ColorScheme__scheme_name_, colordict=None, **colormap)  
  
    copy (name=None)  
        Return a full copy of the object, optionally renaming it.
```

ColorSchemeTable

```
class IPython.utils.coloransi.ColorSchemeTable (scheme_list=None,          de-  
                                                fault_scheme='')  
  
    Bases: dict  
  
    General class to handle tables of color schemes.  
  
    It's basically a dict of color schemes with a couple of shorthand attributes and some convenient meth-  
    ods.  
  
    active_scheme_name -> obvious active_colors -> actual color table of the active scheme  
  
    __init__ (scheme_list=None, default_scheme='')  
        Create a table of color schemes.  
  
        The table can be created empty and manually filled or it can be created with a list of valid color  
        schemes AND the specification for the default active scheme.  
  
    add_scheme (new_scheme)  
        Add a new color scheme to the table.  
  
    clear  
        D.clear() -> None. Remove all items from D.  
  
    copy ()  
        Return full copy of object  
  
    static fromkeys ()  
        dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.  
  
    get  
        D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.  
  
    has_key  
        D.has_key(k) -> True if D has a key k, else False  
  
    items  
        D.items() -> list of D's (key, value) pairs, as 2-tuples  
  
    iteritems  
        D.iteritems() -> an iterator over the (key, value) items of D
```

iterkeys

D.iterkeys() -> an iterator over the keys of D

itervalues

D.itervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

set_active_scheme (*scheme, case_sensitive=0*)

Set the currently active scheme.

Names are by default compared in a case-insensitive way, but this can be changed by setting the parameter `case_sensitive` to true.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does: for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

InputTermColors**class IPython.utils.coloransi.InputTermColors**

Color escape sequences for input prompts.

This class is similar to TermColors, but the escapes are wrapped in “ and ” so that readline can properly know the length of each line and can wrap lines accordingly. Use this class for any colored text which needs to be used in input prompts, such as in calls to `raw_input()`.

This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a NoColor escape which is just the null string, suitable for defining ‘dummy’ color schemes in terminals which get confused by color escapes.

This class should be used as a mixin for building color schemes.

TermColors

class IPython.utils.coloransi.TermColors

Color escape sequences.

This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a NoColor escape which is just the null string, suitable for defining ‘dummy’ color schemes in terminals which get confused by color escapes.

This class should be used as a mixin for building color schemes.

8.104.3 Function

IPython.utils.coloransi.make_color_table(*in_class*)

Build a set of color attributes in a class.

Helper function for building the *TermColors classes.

8.105 utils.data

8.105.1 Module: utils.data

Utilities for working with data structures like lists, dicts and tuples.

8.105.2 Functions

IPython.utils.data.chop(*seq, size*)

Chop a sequence into chunks of the given size.

IPython.utils.data.flatten(*seq*)

Flatten a list of lists (NOT recursive, only works for 2d lists).

IPython.utils.data.get_slice(*seq, start=0, stop=None, step=1*)

Get a slice of a sequence with variable step. Specify start,stop,step.

IPython.utils.data.list2dict(*lst*)

Takes a list of (key,value) pairs and turns it into a dict.

IPython.utils.data.list2dict2(*lst, default=''*)

Takes a list and turns it into a dict. Much slower than list2dict, but more versatile. This version can take lists with sublists of arbitrary length (including sclars).

IPython.utils.data.sort_compare(*lst1, lst2, inplace=1*)

Sort and compare two lists.

By default it does it in place, thus modifying the lists. Use inplace = 0 to avoid that (at the cost of temporary copy creation).


```
IPython.utils.data.uniq_stable(elems)
    uniq_stable(elems) -> list
```

Return from an iterable, a list of all the unique elements in the input, but maintaining the order in which they first appear.

A naive solution to this problem which just makes a dictionary with the elements as keys fails to respect the stability condition, since dictionaries are unsorted by nature.

Note: All elements in the input must be valid dictionary keys for this routine to work, as it internally uses a dictionary for efficiency reasons.

8.106 utils.decorators

8.106.1 Module: `utils.decorators`

Decorators that don't go anywhere else.

This module contains misc. decorators that don't really go with another module in `IPython.utils`. Before putting something here please see if it should go into another topical module in `IPython.utils`.

```
IPython.utils.decorators.flag_calls(func)
    Wrap a function to detect and flag when it gets called.
```

This is a decorator which takes a function and wraps it in a function with a 'called' attribute. `wrapper.called` is initialized to False.

The `wrapper.called` attribute is set to False right before each call to the wrapped function, so if the call fails it remains False. After the call completes, `wrapper.called` is set to True and the output is returned.

Testing for truth in `wrapper.called` allows you to determine if a call to `func()` was attempted and succeeded.

8.107 utils.dir2

8.107.1 Module: `utils.dir2`

A fancy version of Python's builtin `dir()` function.

8.107.2 Functions

```
IPython.utils.dir2.dir2(obj)
    dir2(obj) -> list of strings
```

Extended version of the Python builtin `dir()`, which does a few extra checks, and supports common objects with unusual internals that confuse `dir()`, such as Traits and PyCrust.

This version is guaranteed to return only a list of true strings, whereas `dir()` returns anything that objects inject into themselves, even if they are later not really valid for attribute access (many extension libraries have such bugs).

```
IPython.utils.dir2.get_class_members(cls)
```

8.108 utils.doctestreload

8.108.1 Module: `utils.doctestreload`

A utility for handling the reloading of doctest.

8.108.2 Functions

```
IPython.utils.doctestreload.dhook_wrap(func, *a, **k)
```

Wrap a function call in a `sys.displayhook` controller.

Returns a wrapper around `func` which calls `func`, with all its arguments and keywords unmodified, using the default `sys.displayhook`. Since IPython modifies `sys.displayhook`, it breaks the behavior of certain systems that rely on the default behavior, notably `doctest`.

```
IPython.utils.doctestreload.doctest_reload()
```

Properly reload `doctest` to reuse it interactively.

This routine:

- imports `doctest` but does NOT reload it (see below).
- resets its global ‘master’ attribute to `None`, so that multiple uses of the module interactively don’t produce cumulative reports.
- Monkeypatches its core test runner method to protect it from IPython’s

modified `displayhook`. `Doctest` expects the default `displayhook` behavior deep down, so our modification breaks it completely. For this reason, a hard monkeypatch seems like a reasonable solution rather than asking users to manually use a different `doctest` runner when under IPython.

Notes

This function *used to* reload `doctest`, but this has been disabled because reloading `doctest` unconditionally can cause massive breakage of other `doctest`-dependent modules already in memory, such as those for IPython’s own testing system. The name wasn’t changed to avoid breaking people’s code, but the reload call isn’t actually made anymore.

8.109 utils.frame

8.109.1 Module: `utils.frame`

Utilities for working with stack frames.

8.109.2 Functions

`IPython.utils.frame.debugx(expr, pre_msg='')`
Print the value of an expression from the caller's frame.

Takes an expression, evaluates it in the caller's frame and prints both the given expression and the resulting value (as well as a debug mark indicating the name of the calling function. The input must be of a form suitable for `eval()`.

An optional message can be passed, which will be prepended to the printed `expr->value` pair.

`IPython.utils.frame.extract_vars(*names, **kw)`
Extract a set of variables by name from another frame.

Parameters

- **names*: strings One or more variable names which will be extracted from the caller's

frame.

Keywords

- *depth*: integer (0) How many frames in the stack to walk when looking for your variables.

Examples:

```
In [2]: def func(x): ...: y = 1 ...: print extract_vars('x','y') ...:
```

```
In [3]: func('hello') {'y': 1, 'x': 'hello'}
```

`IPython.utils.frame.extract_vars_above(*names)`
Extract a set of variables by name from another frame.

Similar to `extractVars()`, but with a specified depth of 1, so that names are extracted exactly from above the caller.

This is simply a convenience function so that the very common case (for us) of skipping exactly 1 frame doesn't have to construct a special dict for keyword passing.

8.110 utils.generics

8.110.1 Module: `utils.generics`

Generic functions for extending IPython.

See <http://cheeseshop.python.org/pypi/simplegeneric>.

8.110.2 Functions

`IPython.utils.generics.complete_object(*args, **kw)`

Custom completer dispatching for python objects.

Parameters `obj`: object

The object to complete.

prev_completions: list

List of attributes discovered so far.

This should return the list of attributes in `obj`. If you only wish to :

add to the attributes already discovered normally, return :

`own_attrs + prev_completions. :`

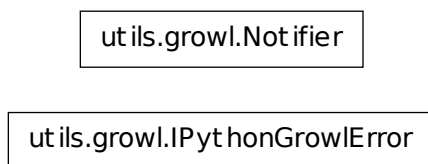
`IPython.utils.generics.inspect_object(*args, **kw)`

Called when you do `obj?`

8.111 `utils.growl`

8.111.1 Module: `utils.growl`

Inheritance diagram for `IPython.utils.growl`:



Utilities using Growl on OS X for notifications.

8.111.2 Classes

`IPythonGrowlError`

`class IPython.utils.growl.IPythonGrowlError`

Bases: `exceptions.Exception`

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
args  
  
message
```

Notifier

```
class IPython.utils.growl.Notifier(app_name)  
    Bases: object  
  
    __init__(app_name)  
  
    notify(title, msg)  
  
    notify_deferred(r, msg)
```

8.111.3 Functions

```
IPython.utils.growl.notify(title, msg)  
IPython.utils.growl.notify_deferred(r, msg)  
IPython.utils.growl.start(app_name)
```

8.112 utils.importstring

8.112.1 Module: `utils.importstring`

A simple utility to import something by its string name.

Authors:

- Brian Granger

```
IPython.utils.importstring.import_item(name)  
    Import and return bar given the string foo.bar.
```

8.113 utils.io

8.113.1 Module: `utils.io`

Inheritance diagram for `IPython.utils.io`:

utils.io.IOStream

utils.io.Tee

utils.io.NLprinter

utils.io.IOTerm

IO related utilities.

8.113.2 Classes

IOStream

class IPython.utils.io.**IOStream**(*stream, fallback*)

__init__(*stream, fallback*)

close()

write(*data*)

IOTerm

class IPython.utils.io.**IOTerm**(*cin=None, cout=None, cerr=None*)

Term holds the file or file-like objects for handling I/O operations.

These are normally just sys.stdin, sys.stdout and sys.stderr but for Windows they can be replaced to allow editing the strings before they are displayed.

__init__(*cin=None, cout=None, cerr=None*)

NLprinter

class IPython.utils.io.**NLprinter**

Print an arbitrarily nested list, indicating index numbers.

An instance of this class called nlprint is available and callable as a function.

`nlprint(list,indent=' ',sep=': ')` -> prints indenting each level by 'indent' and using 'sep' to separate the index from the value.

`__init__()`

Tee

class `IPython.utils.io.Tee` (*file_or_name*, *mode=None*, *channel='stdout'*)

Bases: `object`

A class to duplicate an output stream to `stdout/err`.

This works in a manner very similar to the Unix 'tee' command.

When the object is closed or deleted, it closes the original file given to it for duplication.

`__init__` (*file_or_name*, *mode=None*, *channel='stdout'*)

Construct a new Tee object.

Parameters **file_or_name** : filename or open filehandle (writable)

File that will be duplicated

mode : optional, valid mode for `open()`.

If a filename was give, open with this mode.

channel : str, one of ['`stdout`', '`stderr`']

`close()`

Close the file and restore the channel.

`flush()`

Flush both channels.

`write` (*data*)

Write data to both channels.

8.113.3 Functions

`IPython.utils.io.ask_yes_no` (*prompt*, *default=None*)

Asks a question and returns a boolean (y/n) answer.

If default is given (one of 'y','n'), it is used if the user input is empty. Otherwise the question is repeated until an answer is given.

An EOF is treated as the default answer. If there is no default, an exception is raised to prevent infinite loops.

Valid answers are: y/yes/n/no (match is not case sensitive).

`IPython.utils.io.file_read` (*filename*)

Read a file and close it. Returns the file source.

`IPython.utils.io.file_readlines(filename)`

Read a file and close it. Returns the file source using `readlines()`.

`IPython.utils.io.raw_input_ext(prompt=' ', ps2='... ')`

Similar to `raw_input()`, but accepts extended lines if input ends with `.`

`IPython.utils.io.raw_input_multi(header=' ', ps1='==> ', ps2='..> ', terminate_str='.')`

Take multiple lines of input.

A list with each line of input as a separate element is returned when a termination string is entered (defaults to a single `.`). Input can also terminate via EOF (`^D` in Unix, `^Z-RET` in Windows).

Lines of input which end in `.` are joined into single entries (and a secondary continuation prompt is issued as long as the user terminates lines with `.`). This allows entering very long strings which are still meant to be treated as single entities.

`IPython.utils.io.raw_print(*args, **kw)`

Raw print to `sys.__stdout__`, otherwise identical interface to `print()`.

`IPython.utils.io.raw_print_err(*args, **kw)`

Raw print to `sys.__stderr__`, otherwise identical interface to `print()`.

`IPython.utils.io.temp_pyfile(src, ext='.py')`

Make a temporary python file, return filename and filehandle.

Parameters `src` : string or list of strings (no need for ending newlines if list)

Source code to be written to the file.

ext : optional, string

Extension for the generated file.

Returns (`filename`, `open filehandle`) :

It is the caller's responsibility to close the open file and unlink it.

8.114 utils.ipstruct

8.114.1 Module: `utils.ipstruct`

Inheritance diagram for `IPython.utils.ipstruct`:

utils.ipstruct.Struct

A dict subclass that supports attribute style access.

Authors:

- Fernando Perez (original)
- Brian Granger (refactoring to a dict subclass)

8.114.2 Struct

class IPython.utils.ipstruct.**Struct** (*args, **kw)

Bases: dict

A dict subclass with attribute style access.

This dict subclass has a few extra features:

- Attribute style access.
- Protection of class members (like keys, items) when using attribute style access.
- The ability to restrict assignment to only existing keys.
- Intelligent merging.
- Overloaded operators.

__init__ (*args, **kw)

Initialize with a dictionary, another Struct, or data.

Parameters args : dict, Struct

Initialize with one dict or Struct

kw : dict

Initialize with key, value pairs.

Examples

```
>>> s = Struct(a=10,b=30)
>>> s.a
10
>>> s.b
30
>>> s2 = Struct(s,c=30)
>>> s2.keys()
['a', 'c', 'b']
```

allow_new_attr (allow=True)

Set whether new attributes can be created in this Struct.

This can be used to catch typos by verifying that the attribute user tries to change already exists in this Struct.

clear

D.clear() -> None. Remove all items from D.

copy()

Return a copy as a Struct.

Examples

```
>>> s = Struct(a=10,b=30)
>>> s2 = s.copy()
>>> s2
{'a': 10, 'b': 30}
>>> type(s2).__name__
'Struct'
```

dict()**static fromkeys()**

dict.fromkeys(S[,v]) -> New dict with keys from S and values equal to v. v defaults to None.

get

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

has_key

D.has_key(k) -> True if D has a key k, else False

hasattr(key)

hasattr function available as a method.

Implemented like has_key.

Examples

```
>>> s = Struct(a=10)
>>> s.hasattr('a')
True
>>> s.hasattr('b')
False
>>> s.hasattr('get')
False
```

items

D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems

D.iteritems() -> an iterator over the (key, value) items of D

iterkeys

D.iterkeys() -> an iterator over the keys of D

intervalues

D.intervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

merge (__loc_data__=None, __Struct__conflict_solve=None, **kw)

Merge two Structs with customizable conflict resolution.

This is similar to `update()`, but much more flexible. First, a dict is made from data+key=value pairs. When merging this dict with the Struct S, the optional dictionary ‘conflict’ is used to decide what to do.

If conflict is not given, the default behavior is to preserve any keys with their current value (the opposite of the `update()` method’s behavior).

Parameters __loc_data : dict, Struct

The data to merge into self

__conflict_solve : dict

The conflict policy dict. The keys are binary functions used to resolve the conflict and the values are lists of strings naming the keys the conflict resolution function applies to. Instead of a list of strings a space separated string can be used, like ‘a b c’.

kw : dict

Additional key, value pairs to merge in

Notes

The `__conflict_solve` dict is a dictionary of binary functions which will be used to solve key conflicts. Here is an example:

```
__conflict_solve = dict(
    func1=['a', 'b', 'c'],
    func2=['d', 'e']
)
```

In this case, the function `func1()` will be used to resolve keys ‘a’, ‘b’ and ‘c’ and the function `func2()` will be used for keys ‘d’ and ‘e’. This could also be written as:

```
__conflict_solve = dict(func1='a b c', func2='d e')
```

These functions will be called for each key they apply to with the form:

```
func1(self['a'], other['a'])
```

The return value is used as the final merged value.

As a convenience, `merge()` provides five (the most commonly needed) pre-defined policies: `preserve`, `update`, `add`, `add_flip` and `add_s`. The easiest explanation is their implementation:

```
preserve = lambda old,new: old
update   = lambda old,new: new
add      = lambda old,new: old + new
add_flip = lambda old,new: new + old # note change of order!
add_s    = lambda old,new: old + ' ' + new # only for str!
```

You can use those four words (as strings) as keys instead of defining them as functions, and the merge method will substitute the appropriate functions for you.

For more complicated conflict resolution policies, you still need to construct your own functions.

Examples

This show the default policy:

```
>>> s = Struct(a=10,b=30)
>>> s2 = Struct(a=20,c=40)
>>> s.merge(s2)
>>> s
{'a': 10, 'c': 40, 'b': 30}
```

Now, show how to specify a conflict dict:

```
>>> s = Struct(a=10,b=30)
>>> s2 = Struct(a=20,b=40)
>>> conflict = {'update': 'a', 'add': 'b'}
>>> s.merge(s2, conflict)
>>> s
{'a': 20, 'b': 70}
```

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does: for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

8.115 utils.jsonutil

8.115.1 Module: `utils.jsonutil`

Utilities to manipulate JSON objects.

`IPython.utils.jsonutil.json_clean(obj)`

Clean an object to ensure it's safe to encode in JSON.

Atomic, immutable objects are returned unmodified. Sets and tuples are converted to lists, lists are copied and dicts are also copied.

Note: dicts whose keys could cause collisions upon encoding (such as a dict with both the number 1 and the string '1' as keys) will cause a `ValueError` to be raised.

Parameters `obj`: any python object

Returns `out`: object

A version of the input which will not cause an encoding error when encoded as JSON. Note that this function does not *encode* its inputs, it simply sanitizes it so that there will be no encoding errors later.

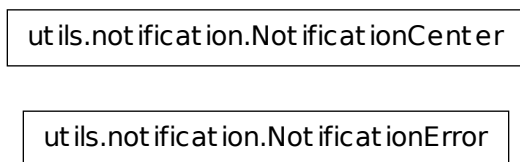
Examples

```
>>> json_clean(4)
4
>>> json_clean(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> json_clean(dict(x=1, y=2))
{'y': 2, 'x': 1}
>>> json_clean(dict(x=1, y=2, z=[1, 2, 3]))
{'y': 2, 'x': 1, 'z': [1, 2, 3]}
>>> json_clean(True)
True
```

8.116 utils.notification

8.116.1 Module: `utils.notification`

Inheritance diagram for `IPython.utils.notification`:



The IPython Core Notification Center.

See `docs/source/development/notification_blueprint.txt` for an overview of the notification module.

Authors:

- Barry Wark
- Brian Granger

8.116.2 Classes

NotificationCenter

class IPython.utils.notification.**NotificationCenter**

Bases: object

Synchronous notification center.

Examples

Here is a simple example of how to use this:

```
import IPython.kernel.core.notification as notification
def callback(ntype, theSender, args={}):
    print ntype,theSender,args

notification.sharedCenter.add_observer(callback, 'NOTIFICATION_TYPE', None)
notification.sharedCenter.post_notification('NOTIFICATION_TYPE', object()) # doctest:+
NOTIFICATION_TYPE ...
```

__init__()

add_observer(*callback*, *ntype*, *sender*)

Add an observer callback to this notification center.

The given callback will be called upon posting of notifications of the given type/sender and will receive any additional arguments passed to `post_notification`.

Parameters **callback** : callable

The callable that will be called by `post_notification()` as “call-back(*ntype*, *sender*, **args*, ***kwargs*)

ntype : hashable

The notification type. If None, all notifications from sender will be posted.

sender : hashable

The notification sender. If None, all notifications of *ntype* will be posted.

post_notification(*ntype*, *sender*, **args*, ***kwargs*)

Post notification to all registered observers.

The registered callback will be called as:

```
callback(ntype, sender, *args, **kwargs)
```

Parameters **ntype** : hashable

The notification type.

sender : hashable

The object sending the notification.

***args** : tuple

The positional arguments to be passed to the callback.

****kwargs** : dict

The keyword argument to be passed to the callback.

Notes

- If no registered observers, performance is $O(1)$.
- Notificaiton order is undefined.
- Notifications are posted synchronously.

remove_all_observers()

Removes all observers from this notification center

NotificationError

class IPython.utils.notification.NotificationError

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

8.117 utils.path

8.117.1 Module: utils.path

Inheritance diagram for IPython.utils.path:

utils.pat h.HomeDirError

Utilities for path handling.

8.117.2 Class

8.117.3 HomeDirError

```
class IPython.utils.path.HomeDirError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

8.117.4 Functions

```
IPython.utils.path.expand_path(s)
    Expand $VARS and ~names in a string, like a shell
```

Examples In [2]: `os.environ['FOO']='test'`

In [3]: `expand_path('variable FOO is $FOO')` Out[3]: `'variable FOO is test'`

```
IPython.utils.path.filefind(filename, path_dirs=None)
    Find a file by looking through a sequence of paths.
```

This iterates through a sequence of paths looking for a file and returns the full, absolute path of the first occurrence of the file. If no set of path dirs is given, the filename is tested as is, after running through `expandvars()` and `expanduser()`. Thus a simple call:

```
filefind('myfile.txt')
```

will find the file in the current working dir, but:

```
filefind('~ /myfile.txt')
```

Will find the file in the users home directory. This function does not automatically try any paths, such as the cwd or the user's home directory.

Parameters `filename` : str

The filename to look for.

`path_dirs` : str, None or sequence of str

The sequence of paths to look for the file in. If None, the filename need to be absolute or be in the cwd. If a string, the string is put into a sequence and the searched. If a sequence, walk through each element and join with `filename`, calling `expandvars()` and `expanduser()` before testing for existence.

Returns Raises :exc:'IOError' or returns absolute path to file. :

`IPython.utils.path.get_home_dir()`

Return the closest possible equivalent to a 'home' directory.

- On POSIX, we try \$HOME.
- On Windows we try: - %HOMESHARE% - %HOMEDRIVE%HOMEPATH% - %USERPROFILE% - Registry hack for My Documents - %HOME%: rare, but some people with unix-like setups may have defined it
- On Dos C:

Currently only Posix and NT are implemented, a HomeDirError exception is raised for all other OSes.

`IPython.utils.path.get_ipython_dir()`

Get the IPython directory for this platform and user.

This uses the logic in `get_home_dir` to find the home directory and the adds .ipython to the end of the path.

`IPython.utils.path.get_ipython_module_path(module_str)`

Find the path to an IPython module in this version of IPython.

This will always find the version of the module that is in this importable IPython package. This will always return the path to the .py version of the module.

`IPython.utils.path.get_ipython_package_dir()`

Get the base directory where IPython itself is installed.

`IPython.utils.path.get_long_path_name(path)`

Expand a path into its long form.

On Windows this expands any ~ in the paths. On other platforms, it is a null operation.

`IPython.utils.path.get_py_filename(name)`

Return a valid python filename in the current directory.

If the given name is not a file, it adds '.py' and searches again. Raises IOError with an informative message if the file isn't found.

`IPython.utils.path.target_outdated(target, deps)`

Determine whether a target is out of date.

`target_outdated(target,deps) -> 1/0`

deps: list of filenames which MUST exist. target: single filename which may or may not exist.

If target doesn't exist or is older than any file listed in deps, return true, otherwise return false.

`IPython.utils.path.target_update(target, deps, cmd)`

Update a target with a given command given a list of dependencies.

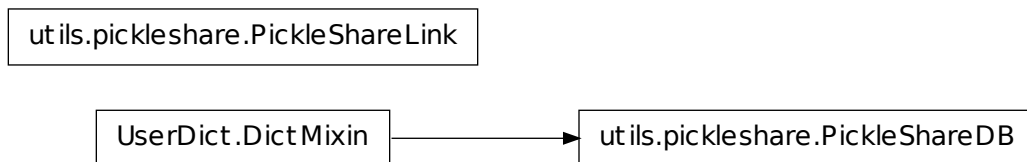
`target_update(target,deps,cmd) -> runs cmd if target is outdated.`

This is just a wrapper around `target_outdated()` which calls the given command if target is outdated.

8.118 utils.pickleshare

8.118.1 Module: `utils.pickleshare`

Inheritance diagram for `IPython.utils.pickleshare`:



PickleShare - a small ‘shelve’ like datastore with concurrency support

Like shelve, a `PickleShareDB` object acts like a normal dictionary. Unlike shelve, many processes can access the database simultaneously. Changing a value in database is immediately visible to other processes accessing the same database.

Concurrency is possible because the values are stored in separate files. Hence the “database” is a directory where *all* files are governed by `PickleShare`.

Example usage:

```
from pickleshare import *
db = PickleShareDB('~/.testpickleshare')
db.clear()
print "Should be empty:", db.items()
db['hello'] = 15
db['aku ankka'] = [1, 2, 313]
db['paths/are/ok/key'] = [1, (5, 46)]
print db.keys()
del db['aku ankka']
```

This module is certainly not ZODB, but can be used for low-load (non-mission-critical) situations where tiny code size trumps the advanced features of a “real” object database.

Installation guide: `easy_install pickleshare`

Author: Ville Vainio <vivainio@gmail.com> License: MIT open source license.

8.118.2 Classes

`PickleShareDB`

```
class IPython.utils.pickleshare.PickleShareDB(root)
    Bases: UserDict.DictMixin
```

The main ‘connection’ object for PickleShare database

__init__ (*root*)

Return a db object that will manage the specied directory

clear ()

get (*key*, *default=None*)

getlink (*folder*)

Get a convenient link for accessing items

has_key (*key*)

hcompress (*hashroot*)

Compress category ‘hashroot’, so hset is fast again

hget will fail if fast_only is True for compressed items (that were hset before hcompress).

hdict (*hashroot*)

Get all data contained in hashed category ‘hashroot’ as dict

hget (*hashroot*, *key*, *default=<object object at 0x10042b810>*, *fast_only=True*)

hashed get

hset (*hashroot*, *key*, *value*)

hashed set

items ()

iteritems ()

iterkeys ()

intervalues ()

keys (*globpat=None*)

All keys in DB, or all keys matching a glob

pop (*key*, **args*)

popitem ()

setdefault (*key*, *default=None*)

uncache (**items*)

Removes all, or specified items from cache

Use this after reading a large amount of large objects to free up memory, when you won’t be needing the objects for a while.

update (*other=None*, ***kwargs*)

values ()

waitget (*key*, *maxwaittime=60*)

Wait (poll) for a key to get a value

Will wait for *maxwaittime* seconds before raising a KeyError. The call exits normally if the *key* field in db gets a value within the timeout period.

Use this for synchronizing different processes or for ensuring that an unfortunately timed “db[‘key’] = newvalue” operation in another process (which causes all ‘get’ operation to cause a KeyError for the duration of pickling) won’t screw up your program logic.

PickleShareLink

class IPython.utils.pickleshare.**PickleShareLink** (*db, keydir*)

A shorthand for accessing nested PickleShare data conveniently.

Created through PickleShareDB.getlink(), example:

```
lnk = db.getlink('myobjects/test')
lnk.foo = 2
lnk.bar = lnk.foo + 5
```

__init__ (*db, keydir*)

8.118.3 Functions

IPython.utils.pickleshare.**gethashfile** (*key*)

IPython.utils.pickleshare.**main** ()

IPython.utils.pickleshare.**stress** ()

IPython.utils.pickleshare.**test** ()

8.119 utils.process

8.119.1 Module: utils.process

Inheritance diagram for IPython.utils.process:

utils.process.FindCmdError

Utilities for working with external processes.

8.119.2 Class

8.119.3 FindCmdError

class IPython.utils.process.**FindCmdError**

Bases: `exceptions.Exception`

__init__()

x.**__init__**(...) initializes x; see x.**__class__**.**__doc__** for signature

args

message

8.119.4 Functions

IPython.utils.process.**abbrev_cwd**()

Return abbreviated version of cwd, e.g. d:mydir

IPython.utils.process.**arg_split**(s, posix=False)

Split a command line's arguments in a shell-like manner.

This is a modified version of the standard library's `shlex.split()` function, but with a default of `posix=False` for splitting, so that quotes in inputs are respected.

IPython.utils.process.**find_cmd**(cmd)

Find absolute path to executable cmd in a cross platform manner.

This function tries to determine the full path to a command line program using *which* on Unix/Linux/OS X and *win32api* on Windows. Most of the time it will use the version that is first on the users *PATH*. If cmd is *python* return *sys.executable*.

Warning, don't use this to find IPython command line programs as there is a risk you will find the wrong one. Instead find those using the following code and looking for the application itself:

```
from IPython.utils.path import get_ipython_module_path
from IPython.utils.process import pycmd2argv
argv = pycmd2argv(get_ipython_module_path('IPython.frontend.terminal.ipapp'))
```

Parameters cmd : str

The command line program to look for.

IPython.utils.process.**pycmd2argv**(cmd)

Take the path of a python command and return a list (argv-style).

This only works on Python based command line programs and will find the location of the python executable using `sys.executable` to make sure the right version is used.

For a given path cmd, this returns [cmd] if cmd's extension is .exe, .com or .bat, and [, cmd] otherwise.

Parameters cmd : string

The path of the command.

Returns argv-style list. :

8.120 utils.strdispatch

8.120.1 Module: `utils.strdispatch`

Inheritance diagram for `IPython.utils.strdispatch`:

`utils.strdispatch.StrDispatch`

String dispatch class to match regexps and dispatch commands.

8.120.2 `StrDispatch`

class `IPython.utils.strdispatch.StrDispatch`

Bases: `object`

Dispatch (lookup) a set of strings / regexps for match.

Example:

```
>>> dis = StrDispatch()
>>> dis.add_s('hei', 34, priority = 4)
>>> dis.add_s('hei', 123, priority = 2)
>>> dis.add_re('h.i', 686)
>>> print list(dis.flat_matches('hei'))
[123, 34, 686]
```

`__init__`()

`add_re`(`regex, obj, priority=0`)

Adds a target regexp for dispatching

`add_s`(`s, obj, priority=0`)

Adds a target 'string' for dispatching

`dispatch`(`key`)

Get a seq of Commandchain objects that match key

`flat_matches`(`key`)

Yield all 'value' targets, without priority

`s_matches`(`key`)

8.121 utils.sysinfo

8.121.1 Module: `utils.sysinfo`

Utilities for getting information about IPython and the system it's running in.

8.121.2 Functions

`IPython.utils.sysinfo.num_cpus()`

Return the effective number of CPUs in the system as an integer.

This cross-platform function makes an attempt at finding the total number of available CPUs in the system, as returned by various underlying system and python calls.

If it can't find a sensible answer, it returns 1 (though an error *may* make it return a large positive number that's actually incorrect).

`IPython.utils.sysinfo.pkg_commit_hash(pkg_path)`

Get short form of commit hash given directory *pkg_path*

There should be a file called 'COMMIT_INFO.txt' in *pkg_path*. This is a file in INI file format, with at least one section: `commit hash`, and two variables `archive_subst_hash` and `install_hash`. The first has a substitution pattern in it which may have been filled by the execution of `git archive` if this is an archive generated that way. The second is filled in by the installation, if the installation is from a git archive.

We get the commit hash from (in order of preference):

- A substituted value in `archive_subst_hash`
- A written commit hash value in `install_hash`
- git output, if we are in a git repository

If all these fail, we return a not-found placeholder tuple

Parameters `pkg_path` : str

directory containing package

Returns `hash_from` : str

Where we got the hash from - description

`hash_str` : str

short form of hash

`IPython.utils.sysinfo.pkg_info(pkg_path)`

Return dict describing the context of this package

Parameters `pkg_path` : str

path containing `__init__.py` for package

Returns `context` : dict

with named parameters of interest

```
IPython.utils.sysinfo.sys_info()
```

Return useful information about IPython and the system, as a string.

8.122 utils.syspathcontext

8.122.1 Module: `utils.syspathcontext`

Inheritance diagram for `IPython.utils.syspathcontext`:

```
utils.syspathcontext.appended_to_syspath
```

```
utils.syspathcontext.prepended_to_syspath
```

Context managers for adding things to `sys.path` temporarily.

Authors:

- Brian Granger

8.122.2 Classes

`appended_to_syspath`

```
class IPython.utils.syspathcontext.appended_to_syspath(dir)
```

Bases: `object`

A context for appending a directory to `sys.path` for a second.

```
__init__(dir)
```

`prepended_to_syspath`

```
class IPython.utils.syspathcontext.prepended_to_syspath(dir)
```

Bases: `object`

A context for prepending a directory to `sys.path` for a second.

```
__init__(dir)
```


8.123 `utils.terminal`

8.123.1 Module: `utils.terminal`

Utilities for working with terminals.

Authors:

- Brian E. Granger
- Fernando Perez
- Alexander Belchenko (e-mail: bialix AT ukr.net)

8.123.2 Functions

`IPython.utils.terminal.freeze_term_title()`

`IPython.utils.terminal.get_terminal_size(defaultx=80, defaulty=25)`

`IPython.utils.terminal.set_term_title(title)`

Set terminal title using the necessary platform-dependent calls.

`IPython.utils.terminal.term_clear()`

`IPython.utils.terminal.toggle_set_term_title(val)`

Control whether `set_term_title` is active or not.

`set_term_title()` allows writing to the console titlebar. In embedded widgets this can cause problems, so this call can be used to toggle it on or off as needed.

The default state of the module is for the function to be disabled.

Parameters `val` : bool

If True, `set_term_title()` actually writes to the terminal (using the appropriate platform-specific module). If False, it is a no-op.

8.124 `utils.text`

8.124.1 Module: `utils.text`

Inheritance diagram for `IPython.utils.text`:

utils.text.LSString

utils.text.SList

Utilities for working with strings and text.

8.124.2 Classes

LSString

class IPython.utils.text.LSString

Bases: str

String derivative with a special access attributes.

These are normal strings, but with the special attributes:

.l (or .list) : value as list (split on newlines). .n (or .nlstr): original value (the string itself).
.s (or .spstr): value as whitespace-separated string. .p (or .paths): list of path objects

Any values which require transformations are computed only once and cached.

Such strings are very useful to efficiently interact with the shell, which typically only understands whitespace-separated options for commands.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

capitalize

S.capitalize() -> string

Return a copy of the string S with only its first character capitalized.

center

S.center(width[, fillchar]) -> string

Return S centered in a string of length width. Padding is done using the specified fill character (default is a space)

count

S.count(sub[, start[, end]]) -> int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode

`S.decode([encoding[,errors]]) -> object`

Decodes `S` using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a `UnicodeDecodeError`. Other possible values are 'ignore' and 'replace' as well as any other name registered with `codecs.register_error` that is able to handle `UnicodeDecodeErrors`.

encode

`S.encode([encoding[,errors]]) -> object`

Encodes `S` using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with `codecs.register_error` that is able to handle `UnicodeEncodeErrors`.

endswith

`S.endswith(suffix[, start[, end]]) -> bool`

Return True if `S` ends with the specified suffix, False otherwise. With optional start, test `S` beginning at that position. With optional end, stop comparing `S` at that position. suffix can also be a tuple of strings to try.

expandtabs

`S.expandtabs([tabsize]) -> string`

Return a copy of `S` where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

find

`S.find(sub [,start [,end]]) -> int`

Return the lowest index in `S` where substring `sub` is found, such that `sub` is contained within `s[start:end]`. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format

`S.format(*args, **kwargs) -> unicode`

get_list()**get_nlstr()****get_paths()****get_spstr()****index**

`S.index(sub [,start [,end]]) -> int`

Like `S.find()` but raise `ValueError` when the substring is not found.

isalnum

`S.isalnum() -> bool`

Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha

S.isalpha() -> bool

Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdigit

S.isdigit() -> bool

Return True if all characters in S are digits and there is at least one character in S, False otherwise.

islower

S.islower() -> bool

Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isspace

S.isspace() -> bool

Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle

S.istitle() -> bool

Return True if S is a titlecased string and there is at least one character in S, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper

S.isupper() -> bool

Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join

S.join(sequence) -> string

Return a string which is the concatenation of the strings in the sequence. The separator between elements is S.

l**list****ljust**

S.ljust(width[, fillchar]) -> string

Return S left-justified in a string of length width. Padding is done using the specified fill character (default is a space).

lower

S.lower() -> string

Return a copy of the string S converted to lowercase.

lstrip

S.lstrip([chars]) -> string or unicode

Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

n**nlstr****p**

partition (sep) -> (head, sep, tail)

Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

paths**replace**

S.replace(old, new[, count]) -> string

Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind

S.rfind(sub [,start [,end]]) -> int

Return the highest index in S where substring sub is found, such that sub is contained within s[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex

S.rindex(sub [,start [,end]]) -> int

Like S.rfind() but raise ValueError when the substring is not found.

rjust

S.rjust(width[, fillchar]) -> string

Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space)

rpartition (sep) -> (tail, sep, head)

Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit

S.rsplit([sep [,maxsplit]]) -> list of strings

Return a list of the words in the string S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator.

rstrip

S.rstrip([chars]) -> string or unicode

Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

s**split**

S.split([sep [,maxsplit]]) -> list of strings

Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines

S.splitlines([keepends]) -> list of strings

Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

spstr**startswith**

S.startswith(prefix[, start[, end]]) -> bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip

S.strip([chars]) -> string or unicode

Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode before stripping

swapcase

S.swapcase() -> string

Return a copy of the string S with uppercase characters converted to lowercase and vice versa.

title

S.title() -> string

Return a titlecased version of S, i.e. words start with uppercase characters, all remaining cased characters have lowercase.

translate

S.translate(table [,deletechars]) -> string

Return a copy of the string S, where all characters occurring in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

upper

S.upper() -> string

Return a copy of the string S converted to uppercase.

zfill

S.zfill(width) -> string

Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

SList

class IPython.utils.text.SList

Bases: list

List derivative with a special access attributes.

These are normal lists, but with the special attributes:

.l (or .list) : value as list (the list itself). .n (or .nlstr): value as a string, joined on newlines.

.s (or .spstr): value as a string, joined on spaces. .p (or .paths): list of path objects

Any values which require transformations are computed only once and cached.

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

append

L.append(object) – append object to end

count

L.count(value) -> integer – return number of occurrences of value

extend

L.extend(iterable) – extend list by appending elements from the iterable

fields (*fields)

Collect whitespace-separated fields from string list

Allows quick awk-like usage of string lists.

Example data (in var a, created by 'a = !ls -l')::

```
-rwxrwxrwx      1 ville None 18 Dec 14 2006 ChangeLog
```

```
drwxrwxrwx+ 6 ville None 0 Oct 24 18:05 IPython
```

a.fields(0) is ['-rwxrwxrwx', 'drwxrwxrwx+'] a.fields(1,0) is ['1 -rwxrwxrwx', '6 drwxrwxrwx+'] (note the joining by space). a.fields(-1) is ['ChangeLog', 'IPython']

IndexErrors are ignored.

Without args, fields() just split()'s the strings.

get_list()

get_nlstr()

get_paths()

get_spstr()

grep(*pattern*, *prune=False*, *field=None*)

Return all strings matching 'pattern' (a regex or callable)

This is case-insensitive. If *prune* is true, return all items NOT matching the pattern.

If *field* is specified, the match must occur in the specified whitespace-separated field.

Examples:

```
a.grep( lambda x: x.startswith('C') )
a.grep('Cha.*log', prune=1)
a.grep('chm', field=-1)
```

index

L.index(value, [start, [stop]]) -> integer – return first index of value. Raises ValueError if the value is not present.

insert

L.insert(index, object) – insert object before index

l

list

n

nlstr

p

paths

pop

L.pop([index]) -> item – remove and return item at index (default last). Raises IndexError if list is empty or index is out of range.

remove

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse

L.reverse() – reverse *IN PLACE*

s

sort (*field=None*, *nums=False*)

sort by specified fields (see fields())

Example:: a.sort(1, nums = True)

Sorts a by second field, in numerical order (so that 21 > 3)

spstr

8.124.3 Functions

`IPython.utils.text.dgrep(pat, *opts)`

Return `grep()` on `dir()+dir(__builtins__)`.

A very common use of `grep()` when working interactively.

`IPython.utils.text.esc_quotes(strng)`

Return the input string with single and double quotes escaped out

`IPython.utils.text.format_screen(strng)`

Format a string for screen printing.

This removes some latex-type format codes.

`IPython.utils.text.grep(pat, list, case=1)`

Simple minded grep-like function. `grep(pat,list)` returns occurrences of `pat` in `list`, `None` on failure.

It only does simple string matching, with no support for regexps. Use the option `case=0` for case-insensitive matching.

`IPython.utils.text.idgrep(pat)`

Case-insensitive `dgrep()`

`IPython.utils.text.igrep(pat, list)`

Synonym for case-insensitive `grep`.

`IPython.utils.text.indent(str, nspaces=4, ntabs=0)`

Indent a string a given number of spaces or tabstops.

`indent(str,nspaces=4,ntabs=0) -> indent str by ntabs+nspaces.`

`IPython.utils.text.list_strings(arg)`

Always return a list of strings, given a string or list of strings as input.

Examples In [7]: `list_strings('A single string')` Out[7]: ['A single string']

In [8]: `list_strings(['A single string in a list'])` Out[8]: ['A single string in a list']

In [9]: `list_strings(['A','list','of','strings'])` Out[9]: ['A', 'list', 'of', 'strings']

`IPython.utils.text.make_quoted_expr(s)`

Return string `s` in appropriate quotes, using raw string if possible.

XXX - example removed because it caused encoding errors in documentation generation. We need a new example that doesn't contain invalid chars.

Note the use of raw string and padding at the end to allow trailing backslash.

`IPython.utils.text.marquee(txt='', width=78, mark='*')`

Return the input string centered in a 'marquee'.

Examples In [16]: `marquee('A test',40)` Out[16]: `***** A test *****`

In [17]: `marquee('A test',40,'-')` Out[17]: `----- A test -----`

In [18]: `marquee('A test',40,' ')` Out[18]: `' A test '`

`IPython.utils.text.native_line_ends (filename, backup=1)`

Convert (in-place) a file to line-ends native to the current OS.

If the optional backup argument is given as false, no backup of the original file is left.

`IPython.utils.text.num_ini_spaces (strng)`

Return the number of initial spaces in a string

`IPython.utils.text.qw (words, flat=0, sep=None, maxsplit=-1)`

Similar to Perl's `qw()` operator, but with some more options.

`qw(words, flat=0, sep=' ', maxsplit=-1) -> words.split(sep, maxsplit)`

`words` can also be a list itself, and with `flat=1`, the output will be recursively flattened.

Examples:

```
>>> qw('1 2')
['1', '2']
```

```
>>> qw(['a b', '1 2', ['m n', 'p q']])
[['a', 'b'], ['1', '2'], [['m', 'n'], ['p', 'q']]]
```

```
>>> qw(['a b', '1 2', ['m n', 'p q']], flat=1)
['a', 'b', '1', '2', 'm', 'n', 'p', 'q']
```

`IPython.utils.text.qw_lol (indata)`

`qw_lol('a b') -> [['a','b']]`, otherwise it's just a call to `qw()`.

We need this to make sure the `modules_` some keys *always* end up as a list of lists.

`IPython.utils.text.qwflat (words, sep=None, maxsplit=-1)`

Calls `qw(words)` in flat mode. It's just a convenient shorthand.

`IPython.utils.text.unquote_ends (istr)`

Remove a single pair of quotes from the endpoints of a string.

8.125 utils.timing

8.125.1 Module: `utils.timing`

Utilities for timing code execution.

8.125.2 Functions

`IPython.utils.timing.timing (func, *args, **kw)`

`timing(func, *args, **kw) -> t_total`

Execute a function once, return the elapsed total CPU time in seconds. This is just the first value in `timings_out()`.

`IPython.utils.timing.timings (reps, func, *args, **kw) -> (t_total, t_per_call)`

Execute a function `reps` times, return a tuple with the elapsed total CPU time in seconds and the time per call. These are just the first two values in `timings_out()`.

`IPython.utils.timing.timings_out (reps, func, *args, **kw) -> (t_total, t_per_call, output)`

Execute a function `reps` times, return a tuple with the elapsed total CPU time in seconds, the time per call and the function's output.

Under Unix, the return value is the sum of user+system time consumed by the process, computed via the `resource` module. This prevents problems related to the wraparound effect which the `time.clock()` function has.

Under Windows the return value is in wall clock seconds. See the documentation for the `time` module for more details.

8.126 `utils.upgradedir`

8.126.1 Module: `utils.upgradedir`

A script/util to upgrade all files in a directory

This is rather conservative in its approach, only copying/overwriting new and unedited files.

To be used by “upgrade” feature.

8.126.2 Functions

`IPython.utils.upgradedir.showdiff (old, new)`

`IPython.utils.upgradedir.upgrade_dir (srcdir, tgt_dir)`

Copy over all files in `srcdir` to `tgt_dir` w/ native line endings

Creates `.upgrade_report` in `tgt_dir` that stores md5sums of all files to notice changed files b/w upgrades.

8.127 `utils.warn`

8.127.1 Module: `utils.warn`

Utilities for warnings. Shouldn't we just use the built in warnings module.

8.127.2 Functions

`IPython.utils.warn.error (msg)`

Equivalent to `warn(msg, level=3)`.

`IPython.utils.warn.fatal (msg, exit_val=1)`

Equivalent to `warn(msg,exit_val=exit_val,level=4)`.

`IPython.utils.warn.info (msg)`

Equivalent to `warn(msg,level=1)`.

`IPython.utils.warn.warn (msg, level=2, exit_val=1)`

Standard warning printer. Gives formatting consistency.

Output is sent to `IPython.utils.io.Term.cerr` (`sys.stderr` by default).

Options:

-level(2): allows finer control: 0 -> Do nothing, dummy function. 1 -> Print message. 2 -> Print 'WARNING:' + message. (Default level). 3 -> Print 'ERROR:' + message. 4 -> Print 'FATAL ERROR:' + message and trigger a `sys.exit(exit_val)`.

-exit_val (1): exit value returned by `sys.exit()` for a level 4 warning. Ignored for all other levels.

8.128 utils.wildcard

8.128.1 Module: `utils.wildcard`

Inheritance diagram for `IPython.utils.wildcard`:

```
graph TD; A[utils.wildcard.NameSpace];
```

Support for wildcard pattern matching in object inspection.

Authors

- Jörgen Stenarson <jorgen.stenarson@bostream.nu>

8.128.2 Class

8.128.3 `NameSpace`

class `IPython.utils.wildcard.NameSpace` (*obj*, *name_pattern='**', *type_pattern='all'*, *ignore_case=True*, *show_all=True*)

Bases: `object`

`NameSpace` holds the dictionary for a namespace and implements filtering on name and types

__init__ (*obj*, *name_pattern*='*', *type_pattern*='all', *ignore_case*=True, *show_all*=True)

filter (*name_pattern*, *type_pattern*)

Return dictionary of filtered namespace.

get_ns ()

Return name space dictionary with objects matching type and name patterns.

get_ns_names ()

Return list of object names in namespace that match the patterns.

ns

Return name space dictionary with objects matching type and name patterns.

ns_names

List of objects in name space that match the type and name patterns.

8.128.4 Functions

IPython.utils.wildcard.**create_typedstr2type_dicts** (*dont_include_in_type2type2str*=['lambda'])

Return dictionaries mapping lower case typename to type objects, from the types package, and vice versa.

IPython.utils.wildcard.**is_type** (*obj*, *typedstr_or_type*)

is_type(*obj*,*typedstr_or_type*) verifies if *obj* is of a certain type or group of types takes strings as parameters of the for 'tuple'<->TupleType 'all' matches all types. TODO: Should be extended for choosing more than one type

IPython.utils.wildcard.**list_namespace** (*namespace*, *type_pattern*, *filter*, *ignore_case*=False, *show_all*=False)

Return dictionary of all objects in namespace that matches *type_pattern* and *filter*.

IPython.utils.wildcard.**show_hidden** (*str*, *show_all*=False)

Return true for strings starting with single _ if *show_all* is true.

FREQUENTLY ASKED QUESTIONS

9.1 General questions

9.2 Questions about parallel computing with IPython

9.2.1 Will IPython speed my Python code up?

Yes and no. When converting a serial code to run in parallel, there often many difficulty questions that need to be answered, such as:

- How should data be decomposed onto the set of processors?
- What are the data movement patterns?
- Can the algorithm be structured to minimize data movement?
- Is dynamic load balancing important?

We can't answer such questions for you. This is the hard (but fun) work of parallel computing. But, once you understand these things IPython will make it easier for you to implement a good solution quickly. Most importantly, you will be able to use the resulting parallel code interactively.

With that said, if your problem is trivial to parallelize, IPython has a number of different interfaces that will enable you to parallelize things is almost no time at all. A good place to start is the `map` method of our `MultiEngineClient`.

9.2.2 What is the best way to use MPI from Python?

9.2.3 What about all the other parallel computing packages in Python?

Some of the unique characteristic of IPython are:

- IPython is the only architecture that abstracts out the notion of a parallel computation in such a way that new models of parallel computing can be explored quickly and easily. If you don't like the models we provide, you can simply create your own using the capabilities we provide.
- IPython is asynchronous from the ground up (we use [Twisted](#)).

- IPython’s architecture is designed to avoid subtle problems that emerge because of Python’s global interpreter lock (GIL).
- While IPython’s architecture is designed to support a wide range of novel parallel computing models, it is fully interoperable with traditional MPI applications.
- IPython has been used and tested extensively on modern supercomputers.
- IPython’s networking layers are completely modular. Thus, is straightforward to replace our existing network protocols with high performance alternatives (ones based upon Myranet/Infiniband).
- IPython is designed from the ground up to support collaborative parallel computing. This enables multiple users to actively develop and run the *same* parallel computation.
- Interactivity is a central goal for us. While IPython does not have to be used interactively, it can be.

9.2.4 Why The IPython controller a bottleneck in my parallel calculation?

A golden rule in parallel computing is that you should only move data around if you absolutely need to. The main reason that the controller becomes a bottleneck is that too much data is being pushed and pulled to and from the engines. If your algorithm is structured in this way, you really should think about alternative ways of handling the data movement. Here are some ideas:

1. Have the engines write data to files on the locals disks of the engines.
2. Have the engines write data to files on a file system that is shared by the engines.
3. Have the engines write data to a database that is shared by the engines.
4. Simply keep data in the persistent memory of the engines and move the computation to the data (rather than the data to the computation).
5. See if you can pass data directly between engines using MPI.

ABOUT IPYTHON

10.1 Credits

IPython was started and continues to be led by Fernando Pérez.

10.1.1 Core developers

As of this writing, core development team consists of the following developers:

- **Fernando Pérez** <Fernando.Perez-AT-berkeley.edu> Project creator and leader, IPython core, parallel computing infrastructure, testing, release manager.
- **Robert Kern** <rkern-AT-enthought.com> Co-mentored the 2005 Google Summer of Code project, work on IPython's core.
- **Brian Granger** <ellisonbg-AT-gmail.com> Parallel computing infrastructure, IPython core.
- **Benjamin (Min) Ragan-Kelley** <benjaminrk-AT-gmail.com> Parallel computing infrastructure.
- **Ville Vainio** <vivainio-AT-gmail.com> IPython core, maintainer of IPython trunk from version 0.7.2 to 0.8.4.
- **Gael Varoquaux** <gael.varoquaux-AT-normalesup.org> wxPython IPython GUI, frontend architecture.
- **Barry Wark** <barrywark-AT-gmail.com> Cocoa GUI, frontend architecture.
- **Laurent Dufrechou** <laurent.dufrechou-AT-gmail.com> wxPython IPython GUI.
- **Jörgen Stenarson** <jorgen.stenarson-AT-bostream.nu> Maintainer of the PyReadline project, which is needed for IPython under windows.

10.1.2 Special thanks

The IPython project is also very grateful to:

Bill Bumgarner <bbum-AT-friday.com>, for providing the DPyGetOpt module that IPython used for parsing command line options through version 0.10.

Ka-Ping Yee <ping-AT-lfw.org>, for providing the Itpl module for convenient and powerful string interpolation with a much nicer syntax than formatting through the ‘%’ operator.

Arnd Baecker <baecker-AT-physik.tu-dresden.de>, for his many very useful suggestions and comments, and lots of help with testing and documentation checking. Many of IPython’s newer features are a result of discussions with him.

Obviously Guido van Rossum and the whole Python development team, for creating a great language for interactive computing.

Fernando would also like to thank Stephen Figgins <fig-AT-monitor.net>, an O’Reilly Python editor. His October 11, 2001 article about IPP and LazyPython, was what got this project started. You can read it at <http://www.onlamp.com/pub/a/python/2001/10/11/pythonnews.html>.

10.1.3 Sponsors

We would like to thank the following entities which, at one point or another, have provided resources and support to IPython:

- Enthought (<http://www.entthought.com>), for hosting IPython’s website and supporting the project in various ways over the years, including significant funding and resources in 2010 for the development of our modern ZeroMQ-based architecture and Qt console frontend.
- Google, for supporting IPython through Summer of Code sponsorships in 2005 and 2010.
- Microsoft Corporation, for funding in 2009 the development of documentation and examples of the Windows HPC Server 2008 support in IPython’s parallel computing tools.
- The Nipy project (<http://nipy.org>) for funding in 2009 a significant refactoring of the entire project codebase that was key.
- Ohio Supercomputer Center (part of Ohio State University Research Foundation) and the Department of Defense High Performance Computing Modernization Program (HPCMP), for sponsoring work in 2009 on the ipcluster script used for starting IPython’s parallel computing processes, as well as the integration between IPython and the Vision environment (<http://mgltools.scripps.edu/packages/vision>). This project would not have been possible without the support and leadership of Jose Unpingco, from Ohio State.
- Tech-X Corporation, for sponsoring a NASA SBIR project in 2008 on IPython’s distributed array and parallel computing capabilities.
- Bivio Software (<http://www.bivio.biz/bp/Intro>), for hosting an IPython sprint in 2006 in addition to their support of the Front Range Pythoneers group in Boulder, CO.

10.1.4 Contributors

And last but not least, all the kind IPython contributors who have contributed new code, bug reports, fixes, comments and ideas. A brief list follows, please let us know if we have omitted your name by accident:

- Mark Voorhies <mark.voorhies-AT-ucsf.edu> Printing support in Qt console.

- Thomas Kluyver <takowl-AT-gmail.com> Port of IPython and its necessary ZeroMQ infrastructure to Python3.
- Evan Patterson <epatters-AT-enthought.com> Qt console frontend with ZeroMQ.
- Justin Riley <justin.t.riley-AT-gmail.com> Contributions to parallel support, Amazon EC2, Sun Grid Engine, documentation.
- Satrajit Ghosh <satra-AT-mit.edu> parallel computing (SGE and much more).
- Thomas Spura <tomspur-AT-fedoraproject.org> various fixes motivated by Fedora support.
- Omar Andrés Zapata Mesa <andresete.chaos-AT-gmail.com> Google Summer of Code 2010, terminal support with ZeroMQ
- Gerardo Gutierrez <muzgash-AT-gmail.com> Google Summer of Code 2010, Qt notebook frontend support with ZeroMQ.
- Paul Ivanov <pivanov314-AT-gmail.com> multiline specials improvements.
- Dav Clark <davclark-AT-berkeley.edu> traitlets improvements.
- David Warde-Farley <dwf-AT-cs.toronto.edu> %timeit fixes.
- Darren Dale <dsdale24-AT-gmail.com>, traits-based configuration system, Qt support.
- Jose Unpingco <unpingco@gmail.com> authored multiple tutorials and screencasts teaching the use of IPython both for interactive and parallel work (available in the documentation part of our website).
- Dan Milstein <danmil-AT-comcast.net> A bold refactor of the core prefilter machinery in the IPython interpreter.
- Jack Moffit <jack-AT-xiph.org> Bug fixes, including the infamous color problem. This bug alone caused many lost hours and frustration, many thanks to him for the fix. I've always been a fan of Ogg & friends, now I have one more reason to like these folks. Jack is also contributing with Debian packaging and many other things.
- Alexander Schmolck <a.schmolck-AT-gmx.net> Emacs work, bug reports, bug fixes, ideas, lots more. The ipython.el mode for (X)Emacs is Alex's code, providing full support for IPython under (X)Emacs.
- Andrea Riciputi <andrea.riciputi-AT-libero.it> Mac OSX information, Fink package management.
- Gary Bishop <gb-AT-cs.unc.edu> Bug reports, and patches to work around the exception handling idiosyncracies of WxPython. Readline and color support for Windows.
- Jeffrey Collins <Jeff.Collins-AT-vexcel.com>. Bug reports. Much improved readline support, including fixes for Python 2.3.
- Dryice Liu <dryice-AT-liu.com.cn> FreeBSD port.
- Mike Heeter <korora-AT-SDF.LONESTAR.ORG>
- Christopher Hart <hart-AT-caltech.edu> PDB integration.
- Milan Zamazal <pdm-AT-zamazal.org> Emacs info.
- Philip Hisley <compsys-AT-starpower.net>
- Holger Krekel <pyth-AT-devel.trillke.net> Tab completion, lots more.

- Robin Siebler <robinsiebler-AT-starband.net>
- Ralf Ahlbrink <ralf_ahlbrink-AT-web.de>
- Thorsten Kampe <thorsten-AT-thorstenkampe.de>
- Fredrik Kant <fredrik.kant-AT-front.com> Windows setup.
- Syver Enstad <syver-en-AT-online.no> Windows setup.
- Richard <rx-AT-renre-europe.com> Global embedding.
- Hayden Callow <h.callow-AT-elec.canterbury.ac.nz> Gnuplot.py 1.6 compatibility.
- Leonardo Santagada <retype-AT-terra.com.br> Fixes for Windows installation.
- Christopher Armstrong <radix-AT-twistedmatrix.com> Bugfixes.
- Francois Pinard <pinard-AT-iro.umontreal.ca> Code and documentation fixes.
- Cory Dodt <cdodt-AT-fcoe.k12.ca.us> Bug reports and Windows ideas. Patches for Windows installer.
- Olivier Aubert <oaubert-AT-bat710.univ-lyon1.fr> New magics.
- King C. Shu <kingshu-AT-myrealbox.com> Autoindent patch.
- Chris Drexler <chris-AT-ac-drexler.de> Readline packages for Win32/CygWin.
- Gustavo Cordova Avila <gcordova-AT-sismex.com> EvalDict code for nice, lightweight string interpolation.
- Kasper Souren <Kasper.Souren-AT-ircam.fr> Bug reports, ideas.
- Gever Tulley <gever-AT-helium.com> Code contributions.
- Ralf Schmitt <ralf-AT-brainbot.com> Bug reports & fixes.
- Oliver Sander <osander-AT-gmx.de> Bug reports.
- Rod Holland <rh-AT-structurelabs.com> Bug reports and fixes to logging module.
- Daniel ‘Dang’ Griffith <pythondev-dang-AT-lazytwinares.net> Fixes, enhancement suggestions for system shell use.
- Viktor Ransmayr <viktor.ransmayr-AT-t-online.de> Tests and reports on Windows installation issues. Contributed a true Windows binary installer.
- Mike Salib <msalib-AT-mit.edu> Help fixing a subtle bug related to traceback printing.
- W.J. van der Laan <gnufnork-AT-hetdigitelegat.nl> Bash-like prompt specials.
- Antoon Pardon <Antoon.Pardon-AT-rece.vub.ac.be> Critical fix for the multithreaded IPython.
- John Hunter <jdhunter-AT-nitace.bsd.uchicago.edu> Matplotlib author, helped with all the development of support for matplotlib in IPython, including making necessary changes to matplotlib itself.
- Matthew Arnison <maffew-AT-cat.org.au> Bug reports, ‘%run -d’ idea.
- Prabhu Ramachandran <prabhu_r-AT-users.sourceforge.net> Help with (X)Emacs support, threading patches, ideas...

- Norbert Tretkowski <tretkowski-AT-inittab.de> help with Debian packaging and distribution.
- George Sakkis <gsakkis-AT-edden.rutgers.edu> New matcher for tab-completing named arguments of user-defined functions.
- Jörgen Stenarson <jorgen.stenarson-AT-bostream.nu> Wildcard support implementation for searching namespaces.
- Vivian De Smedt <vivian-AT-vdesmedt.com> Debugger enhancements, so that when pdb is activated from within IPython, coloring, tab completion and other features continue to work seamlessly.
- Scott Tsai <scottt958-AT-yahoo.com.tw> Support for automatic editor invocation on syntax errors (see <http://www.scipy.net/roundup/ipython/issue36>).
- Alexander Belchenko <bialix-AT-ukr.net> Improvements for win32 paging system.
- Will Maier <willmaier-AT-ml1.net> Official OpenBSD port.
- Ondrej Certik <ondrej-AT-certik.cz> Set up the IPython docs to use the new Sphinx system used by Python, Matplotlib and many more projects.
- Stefan van der Walt <stefan-AT-sun.ac.za> Design and prototype of the Traits based config system.

10.2 History

10.2.1 Origins

IPython was starting in 2001 by Fernando Perez while he was a graduate student at the University of Colorado, Boulder. IPython as we know it today grew out of the following three projects:

- ipython by Fernando Pérez. Fernando began using Python and ipython began as an outgrowth of his desire for things like Mathematica-style prompts, access to previous output (again like Mathematica's `%` syntax) and a flexible configuration system (something better than `PYTHONSTARTUP`).
- IPP by Janko Hauser. Very well organized, great usability. Had an old help system. IPP was used as the “container” code into which Fernando added the functionality from ipython and LazyPython.
- LazyPython by Nathan Gray. Simple but very powerful. The quick syntax (auto parens, auto quotes) and verbose/colored tracebacks were all taken from here.

Here is how Fernando describes the early history of IPython:

When I found out about IPP and LazyPython I tried to join all three into a unified system. I thought this could provide a very nice working environment, both for regular programming and scientific computing: shell-like features, IDL/Matlab numerics, Mathematica-type prompt history and great object introspection and help facilities. I think it worked reasonably well, though it was a lot more work than I had initially planned.

10.3 License and Copyright

10.3.1 License

IPython is licensed under the terms of the new or revised BSD license, as follows:

```
Copyright (c) 2008, IPython Development Team
```

```
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions are  
met:
```

```
Redistributions of source code must retain the above copyright notice,  
this list of conditions and the following disclaimer.
```

```
Redistributions in binary form must reproduce the above copyright notice,  
this list of conditions and the following disclaimer in the documentation  
and/or other materials provided with the distribution.
```

```
Neither the name of the IPython Development Team nor the names of its  
contributors may be used to endorse or promote products derived from this  
software without specific prior written permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS  
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,  
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR  
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

10.3.2 About the IPython Development Team

Fernando Perez began IPython in 2001 based on code from Janko Hauser <jhauser-AT-zscout.de> and Nathaniel Gray <n8gray-AT-caltech.edu>. Fernando is still the project lead.

The IPython Development Team is the set of all contributors to the IPython project. This includes all of the IPython subprojects. Here is a list of the currently active contributors:

- Matthieu Brucher
- Ondrej Certik
- Laurent Dufrechou
- Robert Kern

- Brian E. Granger
- Fernando Perez (project leader)
- Benjamin Ragan-Kelley
- Ville M. Vainio
- Gael Varoquaux
- Stefan van der Walt
- Barry Wark

If your name is missing, please add it.

10.3.3 Our Copyright Policy

IPython uses a shared copyright model. Each contributor maintains copyright over their contributions to IPython. But, it is important to note that these contributions are typically only changes (diffs/commits) to the repositories. Thus, the IPython source code, in its entirety is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire IPython Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the IPython repositories.

Any new code contributed to IPython must be licensed under the BSD license or a similar (MIT) open source license.

10.3.4 Miscellaneous

Some files (DPyGetOpt.py, for example) may be licensed under different conditions. Ultimately each file indicates clearly the conditions under which its author/authors have decided to publish the code.

Versions of IPython up to and including 0.6.3 were released under the GNU Lesser General Public License (LGPL), available at <http://www.gnu.org/copyleft/lesser.html>.

BIBLIOGRAPHY

- [Twisted] Twisted matrix. <http://twistedmatrix.org>
- [ZopeInterface] <http://pypi.python.org/pypi/zope.interface>
- [Foolscap] Foolscap network protocol. <http://foolscap.lothar.com/trac>
- [pyOpenSSL] pyOpenSSL. <http://pyopenssl.sourceforge.net>
- [Matplotlib] Matplotlib. <http://matplotlib.sourceforge.net>
- [PyQt] PyQt4 <http://www.riverbankcomputing.co.uk/software/pyqt/download>
- [pygments] Pygments <http://pygments.org/>
- [Capability] Capability-based security, http://en.wikipedia.org/wiki/Capability-based_security
- [PBS] Portable Batch System. <http://www.openpbs.org/>
- [SSH] SSH-Agent <http://en.wikipedia.org/wiki/Ssh-agent>
- [MPI] Message Passing Interface. <http://www-unix.mcs.anl.gov/mpi/>
- [mpi4py] MPI for Python. mpi4py: <http://mpi4py.scipy.org/>
- [OpenMPI] Open MPI. <http://www.open-mpi.org/>
- [PyTrilinos] PyTrilinos. <http://trilinos.sandia.gov/packages/pytrilinos/>
- [RFC5246] <<http://tools.ietf.org/html/rfc5246>>
- [Emacs] Emacs. <http://www.gnu.org/software/emacs/>
- [TextMate] TextMate: the missing editor. <http://macromates.com/>
- [vim] vim. <http://www.vim.org/>
- [Git] The Git version control system.
- [Github.com] Github.com. <http://github.com>
- [PEP8] Python Enhancement Proposal 8. <http://www.python.org/peps/pep-0008.html>
- [reStructuredText] reStructuredText. <http://docutils.sourceforge.net/rst.html>
- [Sphinx] Sphinx. <http://sphinx.pocoo.org/>

[MatplotlibDocGuide] http://matplotlib.sourceforge.net/devel/documenting_mpl.html

[PEP257] PEP 257. <http://www.python.org/peps/pep-0257.html>

[NumPyDocGuide] NumPy documentation guide. <http://projects.scipy.org/numpy/wiki/CodingStyleGuidelines>

[NumPyExampleDocstring] NumPy example docstring. http://projects.scipy.org/numpy/browser/trunk/doc/EXAMPLE_DOCSTRING.rst

PYTHON MODULE INDEX

i

IPython.config.loader, 203
IPython.core.alias, 209
IPython.core.application, 212
IPython.core.autocall, 217
IPython.core.builtin_trap, 218
IPython.core.compilerop, 220
IPython.core.completer, 221
IPython.core.completerlib, 226
IPython.core.crashhandler, 227
IPython.core.debugger, 229
IPython.core.display, 237
IPython.core.display_trap, 239
IPython.core.displayhook, 240
IPython.core.displaypub, 243
IPython.core.error, 246
IPython.core.excolors, 248
IPython.core.extensions, 248
IPython.core.formatters, 251
IPython.core.history, 271
IPython.core.hooks, 275
IPython.core.inputsplitter, 277
IPython.core.interactiveshell, 284
IPython.core.ipapi, 324
IPython.core.logger, 324
IPython.core.macro, 325
IPython.core.magic, 326
IPython.core.oinspect, 348
IPython.core.page, 353
IPython.core.payload, 355
IPython.core.payloadpage, 356
IPython.core.plugin, 357
IPython.core.prefilter, 361
IPython.core.prompts, 396
IPython.core.splitinput, 399
IPython.core.ultratb, 399
IPython.kernel.clientconnector, 411
IPython.kernel.clientinterfaces, 420
IPython.kernel.codeutil, 425
IPython.kernel.controllerservice, 426
IPython.kernel.core.display_formatter, 432
IPython.kernel.core.display_trap, 433
IPython.kernel.core.error, 434
IPython.kernel.core.fd_redirector, 436
IPython.kernel.core.file_like, 437
IPython.kernel.core.history, 438
IPython.kernel.core.interpreter, 440
IPython.kernel.core.macro, 444
IPython.kernel.core.magic, 445
IPython.kernel.core.message_cache, 446
IPython.kernel.core.output_trap, 447
IPython.kernel.core.prompts, 448
IPython.kernel.core.redirector_output_trap, 452
IPython.kernel.core.sync_traceback_trap, 453
IPython.kernel.core.traceback_formatter, 454
IPython.kernel.core.traceback_trap, 455
IPython.kernel.core.util, 456
IPython.kernel.engineconnector, 459
IPython.kernel.enginefc, 460
IPython.kernel.engineservice, 468
IPython.kernel.error, 490
IPython.kernel.ipclusterapp, 498
IPython.kernel.ipengineapp, 501
IPython.kernel.map, 505

[IPython.kernel.mapper](#), 506
[IPython.kernel.multiengine](#), 515
[IPython.kernel.multiengineclient](#), 543
[IPython.kernel.multienginefc](#), 556
[IPython.kernel.newserialized](#), 563
[IPython.kernel.parallelfunction](#), 569
[IPython.kernel.pbutil](#), 577
[IPython.kernel.pendingdeferred](#), 578
[IPython.kernel.pickleutil](#), 579
[IPython.kernel.twistedutil](#), 580
[IPython.kernel.util](#), 584
[IPython.lib.backgroundjobs](#), 585
[IPython.lib.clipboard](#), 589
[IPython.lib.deepreload](#), 590
[IPython.lib.demo](#), 591
[IPython.lib.guisupport](#), 604
[IPython.lib.inputhook](#), 606
[IPython.lib.inputhookwx](#), 609
[IPython.lib.irunner](#), 613
[IPython.lib.latextools](#), 618
[IPython.lib.pylabtools](#), 619
[IPython.testing](#), 621
[IPython.testing.decorators](#), 621
[IPython.testing.decorators_trial](#), 623
[IPython.testing.globalipapp](#), 624
[IPython.testing.ipctest](#), 626
[IPython.testing.ipunittest](#), 628
[IPython.testing.mkdoctests](#), 630
[IPython.testing.nosepatch](#), 631
[IPython.testing.parametric](#), 632
[IPython.testing.plugin.dtexample](#), 632
[IPython.testing.plugin.show_refs](#), 634
[IPython.testing.plugin.simple](#), 635
[IPython.testing.plugin.test_ipdoctest](#), 635
[IPython.testing.plugin.test_refs](#), 636
[IPython.testing.tools](#), 637
[IPython.utils.attic](#), 641
[IPython.utils.autoattr](#), 643
[IPython.utils.coloransi](#), 645
[IPython.utils.data](#), 648
[IPython.utils.decorators](#), 649
[IPython.utils.dir2](#), 649
[IPython.utils.doctestreload](#), 650
[IPython.utils.frame](#), 651
[IPython.utils.generics](#), 651
[IPython.utils.growl](#), 652
[IPython.utils.importstring](#), 653
[IPython.utils.io](#), 654
[IPython.utils.ipstruct](#), 656
[IPython.utils.jsonutil](#), 660
[IPython.utils.notification](#), 661
[IPython.utils.path](#), 663
[IPython.utils.pickleshare](#), 666
[IPython.utils.process](#), 668
[IPython.utils.PyColorize](#), 640
[IPython.utils.strdispatch](#), 670
[IPython.utils.sysinfo](#), 671
[IPython.utils.syspathcontext](#), 672
[IPython.utils.terminal](#), 673
[IPython.utils.text](#), 674
[IPython.utils.timing](#), 682
[IPython.utils.upgradedir](#), 683
[IPython.utils.warn](#), 683
[IPython.utils.wildcard](#), 684

INDEX

Symbols

%PATH%, 105

__init__() (IPython.config.loader.ArgParseConfigLoader method), 203

__init__() (IPython.config.loader.ArgumentParser method), 204

__init__() (IPython.config.loader.CommandLineConfigLoader method), 205

__init__() (IPython.config.loader.Config method), 206

__init__() (IPython.config.loader.ConfigError method), 207

__init__() (IPython.config.loader.ConfigLoader method), 207

__init__() (IPython.config.loader.ConfigLoaderError method), 208

__init__() (IPython.config.loader.FileConfigLoader method), 208

__init__() (IPython.config.loader.PyFileConfigLoader method), 209

__init__() (IPython.core.alias.AliasError method), 209

__init__() (IPython.core.alias.AliasManager method), 210

__init__() (IPython.core.alias.InvalidAliasError method), 212

__init__() (IPython.core.application.Application method), 213

__init__() (IPython.core.application.ApplicationError method), 216

__init__() (IPython.core.application.BaseAppConfigLoader method), 216

__init__() (IPython.core.autocall.IPyAutocall method), 217

__init__() (IPython.core.builtin_trap.BuiltinTrap method), 218

__init__() (IPython.core.compilerop.CachingCompiler method), 220

__init__() (IPython.core.completer.Bunch method), 222

__init__() (IPython.core.completer.Completer method), 222

__init__() (IPython.core.completer.CompletionSplitter method), 223

__init__() (IPython.core.completer.IPCompleter method), 223

__init__() (IPython.core.crashhandler.CrashHandler method), 228

__init__() (IPython.core.debugger.Pdb method), 229

__init__() (IPython.core.debugger.Tracer method), 237

__init__() (IPython.core.display_trap.DisplayTrap method), 239

__init__() (IPython.core.displayhook.DisplayHook method), 241

__init__() (IPython.core.displaypub.DisplayPublisher method), 244

__init__() (IPython.core.error.IPythonCoreError method), 247

__init__() (IPython.core.error.TryNext method), 247

__init__() (IPython.core.error.UsageError method), 247

__init__() (IPython.core.extensions.ExtensionManager method), 249

__init__() (IPython.core.formatters.BaseFormatter method), 251

__init__() (IPython.core.formatters.DisplayFormatter method), 254

__init__() (IPython.core.formatters.FormatterABC method), 256

__init__() (IPython.core.formatters.HTMLFormatter method), 256

__init__() (IPython.core.formatters.JSONFormatter

method), 258

`__init__()` (IPython.core.formatters.LatexFormatter method), 261

`__init__()` (IPython.core.formatters.PNGFormatter method), 263

`__init__()` (IPython.core.formatters.PlainTextFormatter method), 265

`__init__()` (IPython.core.formatters.SVGFormatter method), 268

`__init__()` (IPython.core.history.HistoryManager method), 271

`__init__()` (IPython.core.history.HistorySaveThread method), 273

`__init__()` (IPython.core.history.ShadowHist method), 273

`__init__()` (IPython.core.hooks.CommandChainDispatcher method), 276

`__init__()` (IPython.core.inputsplitter.EscapedTransformer method), 278

`__init__()` (IPython.core.inputsplitter.IPythonInputSplitter method), 278

`__init__()` (IPython.core.inputsplitter.InputSplitter method), 280

`__init__()` (IPython.core.inputsplitter.LineInfo method), 282

`__init__()` (IPython.core.interactiveshell.InteractiveShell method), 285

`__init__()` (IPython.core.interactiveshell.InteractiveShellABC method), 322

`__init__()` (IPython.core.interactiveshell.MultipleInstanceError method), 322

`__init__()` (IPython.core.interactiveshell.SeparateStr method), 322

`__init__()` (IPython.core.interactiveshell.SpaceInInput method), 323

`__init__()` (IPython.core.logger.Logger method), 324

`__init__()` (IPython.core.macro.Macro method), 326

`__init__()` (IPython.core.magic.Magic method), 327

`__init__()` (IPython.core.oinspect.Inspector method), 349

`__init__()` (IPython.core.oinspect.myStringIO method), 351

`__init__()` (IPython.core.payload.PayloadManager method), 355

`__init__()` (IPython.core.plugin.Plugin method), 358

`__init__()` (IPython.core.plugin.PluginManager method), 359

`__init__()` (IPython.core.prefilter.AliasChecker method), 362

`__init__()` (IPython.core.prefilter.AliasHandler method), 363

`__init__()` (IPython.core.prefilter.AssignMagicTransformer method), 364

`__init__()` (IPython.core.prefilter.AssignSystemTransformer method), 366

`__init__()` (IPython.core.prefilter.AssignmentChecker method), 367

`__init__()` (IPython.core.prefilter.AutoHandler method), 368

`__init__()` (IPython.core.prefilter.AutoMagicChecker method), 369

`__init__()` (IPython.core.prefilter.AutocallChecker method), 371

`__init__()` (IPython.core.prefilter.EmacsChecker method), 372

`__init__()` (IPython.core.prefilter.EmacsHandler method), 373

`__init__()` (IPython.core.prefilter.EscCharsChecker method), 375

`__init__()` (IPython.core.prefilter.HelpHandler method), 376

`__init__()` (IPython.core.prefilter.IPyAutocallChecker method), 377

`__init__()` (IPython.core.prefilter.IPyPromptTransformer method), 379

`__init__()` (IPython.core.prefilter.LineInfo method), 380

`__init__()` (IPython.core.prefilter.MagicHandler method), 380

`__init__()` (IPython.core.prefilter.MultiLineMagicChecker method), 382

`__init__()` (IPython.core.prefilter.PrefilterChecker method), 383

`__init__()` (IPython.core.prefilter.PrefilterError method), 384

`__init__()` (IPython.core.prefilter.PrefilterHandler method), 385

`__init__()` (IPython.core.prefilter.PrefilterManager method), 386

`__init__()` (IPython.core.prefilter.PrefilterTransformer method), 389

`__init__()` (IPython.core.prefilter.PyPromptTransformer method), 390

`__init__()` (IPython.core.prefilter.PythonOpsChecker method), 392

<code>__init__()</code> (IPython.core.prefilter.ShellEscapeChecker method), 393	<code>__init__()</code> (IPython.kernel.controllerservice.IControllerCore class method), 430
<code>__init__()</code> (IPython.core.prefilter.ShellEscapeHandler method), 394	<code>__init__()</code> (IPython.kernel.core.display_formatter.IDisplayFormatter method), 432
<code>__init__()</code> (IPython.core.prompts.BasePrompt method), 396	<code>__init__()</code> (IPython.kernel.core.display_formatter.PPrintDisplayFormatter method), 433
<code>__init__()</code> (IPython.core.prompts.Prompt1 method), 397	<code>__init__()</code> (IPython.kernel.core.display_formatter.ReprDisplayFormatter method), 433
<code>__init__()</code> (IPython.core.prompts.Prompt2 method), 397	<code>__init__()</code> (IPython.kernel.core.display_trap.DisplayTrap method), 434
<code>__init__()</code> (IPython.core.prompts.PromptOutput method), 398	<code>__init__()</code> (IPython.kernel.core.error.ControllerCreationError method), 434
<code>__init__()</code> (IPython.core.ultratb.AutoFormattedTB method), 401	<code>__init__()</code> (IPython.kernel.core.error.ControllerError method), 435
<code>__init__()</code> (IPython.core.ultratb.ColorTB method), 402	<code>__init__()</code> (IPython.kernel.core.error.EngineCreationError method), 435
<code>__init__()</code> (IPython.core.ultratb.FormattedTB method), 404	<code>__init__()</code> (IPython.kernel.core.error.EngineError method), 435
<code>__init__()</code> (IPython.core.ultratb.ListTB method), 406	<code>__init__()</code> (IPython.kernel.core.error.IPythonError method), 435
<code>__init__()</code> (IPython.core.ultratb.SyntaxTB method), 407	<code>__init__()</code> (IPython.kernel.core.fd_redirector.FDRedirector method), 436
<code>__init__()</code> (IPython.core.ultratb.TBTools method), 408	<code>__init__()</code> (IPython.kernel.core.file_like.FileLike method), 437
<code>__init__()</code> (IPython.core.ultratb.VerboseTB method), 409	<code>__init__()</code> (IPython.kernel.core.history.FrontEndHistory method), 438
<code>__init__()</code> (IPython.kernel.clientconnector.AsyncClientConnector method), 411	<code>__init__()</code> (IPython.kernel.core.history.History method), 438
<code>__init__()</code> (IPython.kernel.clientconnector.AsyncClusterConnector method), 414	<code>__init__()</code> (IPython.kernel.core.history.InterpreterHistory method), 439
<code>__init__()</code> (IPython.kernel.clientconnector.ClientConnector method), 415	<code>__init__()</code> (IPython.kernel.core.interpreter.Interpreter method), 440
<code>__init__()</code> (IPython.kernel.clientconnector.ClientConnectorError method), 417	<code>__init__()</code> (IPython.kernel.core.interpreter.NotDefined method), 444
<code>__init__()</code> (IPython.kernel.clientconnector.Cluster method), 417	<code>__init__()</code> (IPython.kernel.core.macro.Macro method), 445
<code>__init__()</code> (IPython.kernel.clientconnector.ClusterStateError method), 420	<code>__init__()</code> (IPython.kernel.core.magic.Magic method), 445
<code>__init__()</code> (IPython.kernel.clientinterfaces.IBlockingClientInterface class method), 420	<code>__init__()</code> (IPython.kernel.core.message_cache.IMessageCache method), 446
<code>__init__()</code> (IPython.kernel.clientinterfaces.IFCCClientInterface class method), 423	<code>__init__()</code> (IPython.kernel.core.message_cache.SimpleMessageCache method), 447
<code>__init__()</code> (IPython.kernel.controllerservice.ControllerAdapterBase method), 426	<code>__init__()</code> (IPython.kernel.core.output_trap.OutputTrap method), 447
<code>__init__()</code> (IPython.kernel.controllerservice.ControllerService method), 427	<code>__init__()</code> (IPython.kernel.core.prompts.BasePrompt method), 449
<code>__init__()</code> (IPython.kernel.controllerservice.IControllerBase class method), 427	<code>__init__()</code> (IPython.kernel.core.prompts.CachedOutput method), 449

<code>__init__()</code> (IPython.kernel.core.prompts.Prompt1 method), 450	<code>__init__()</code> (IPython.kernel.engineservice.IEngineSerialized class method), 480
<code>__init__()</code> (IPython.kernel.core.prompts.Prompt2 method), 450	<code>__init__()</code> (IPython.kernel.engineservice.IEngineThreaded class method), 482
<code>__init__()</code> (IPython.kernel.core.prompts.PromptOut method), 451	<code>__init__()</code> (IPython.kernel.engineservice.QueuedEngine method), 485
<code>__init__()</code> (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), 452	<code>__init__()</code> (IPython.kernel.engineservice.StrictDict method), 487
<code>__init__()</code> (IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method), 453	<code>__init__()</code> (IPython.kernel.engineservice.ThreadedEngineService method), 488
<code>__init__()</code> (IPython.kernel.core.traceback_formatter.ITracebackFormatter method), 454	<code>__init__()</code> (IPython.kernel.error.AbortedPendingDeferredError method), 490
<code>__init__()</code> (IPython.kernel.core.traceback_formatter.PlainTracebackFormatter method), 455	<code>__init__()</code> (IPython.kernel.error.ClientError method), 491
<code>__init__()</code> (IPython.kernel.core.traceback_trap.TracebackTrap method), 455	<code>__init__()</code> (IPython.kernel.error.CompositeError method), 491
<code>__init__()</code> (IPython.kernel.core.util.Bunch method), 456	<code>__init__()</code> (IPython.kernel.error.ConnectionError method), 491
<code>__init__()</code> (IPython.kernel.core.util.InputList method), 457	<code>__init__()</code> (IPython.kernel.error.FileTimeoutError method), 491
<code>__init__()</code> (IPython.kernel.engineconnector.EngineConnector method), 460	<code>__init__()</code> (IPython.kernel.error.IdInUse method), 492
<code>__init__()</code> (IPython.kernel.engineconnector.EngineConnectorError method), 460	<code>__init__()</code> (IPython.kernel.error.InvalidClientID method), 492
<code>__init__()</code> (IPython.kernel.enginefc.EngineFromReferenceFromService method), 461	<code>__init__()</code> (IPython.kernel.error.InvalidDeferredID method), 492
<code>__init__()</code> (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	<code>__init__()</code> (IPython.kernel.error.InvalidEngineID method), 492
<code>__init__()</code> (IPython.kernel.enginefc.FCRemoteEngineReferenceFromService method), 463	<code>__init__()</code> (IPython.kernel.error.InvalidProperty method), 493
<code>__init__()</code> (IPython.kernel.enginefc.IFCControllerBase class method), 463	<code>__init__()</code> (IPython.kernel.error.KernelError method), 493
<code>__init__()</code> (IPython.kernel.enginefc.IFCEngine class method), 466	<code>__init__()</code> (IPython.kernel.error.MessageSizeError method), 493
<code>__init__()</code> (IPython.kernel.engineservice.Command method), 469	<code>__init__()</code> (IPython.kernel.error.MissingBlockArgument method), 493
<code>__init__()</code> (IPython.kernel.engineservice.EngineAPI method), 469	<code>__init__()</code> (IPython.kernel.error.NoEnginesRegistered method), 494
<code>__init__()</code> (IPython.kernel.engineservice.EngineService method), 469	<code>__init__()</code> (IPython.kernel.error.NotAPendingResult method), 494
<code>__init__()</code> (IPython.kernel.engineservice.IEngineBase class method), 471	<code>__init__()</code> (IPython.kernel.error.NotDefined method), 494
<code>__init__()</code> (IPython.kernel.engineservice.IEngineCore class method), 473	<code>__init__()</code> (IPython.kernel.error.PBMessageSizeError method), 494
<code>__init__()</code> (IPython.kernel.engineservice.IEngineProperties class method), 475	<code>__init__()</code> (IPython.kernel.error.ProtocolError method), 494
<code>__init__()</code> (IPython.kernel.engineservice.IEngineQueued class method), 478	<code>__init__()</code> (IPython.kernel.error.QueueCleared method), 495

[__init__\(\) \(IPython.kernel.error.ResultAlreadyRetrieved class method\), 495](#)
[__init__\(\) \(IPython.kernel.error.ResultNotCompleted class method\), 495](#)
[__init__\(\) \(IPython.kernel.error.SecurityError class method\), 495](#)
[__init__\(\) \(IPython.kernel.error.SerializationError class method\), 496](#)
[__init__\(\) \(IPython.kernel.error.StopLocalExecution class method\), 496](#)
[__init__\(\) \(IPython.kernel.error.TaskAborted class method\), 496](#)
[__init__\(\) \(IPython.kernel.error.TaskRejectError class method\), 497](#)
[__init__\(\) \(IPython.kernel.error.TaskTimeout class method\), 497](#)
[__init__\(\) \(IPython.kernel.error.UnpickleableException class method\), 497](#)
[__init__\(\) \(IPython.kernel.ipclusterapp.IPClusterApp class method\), 498](#)
[__init__\(\) \(IPython.kernel.ipclusterapp.IPClusterAppConfigLoader class method\), 500](#)
[__init__\(\) \(IPython.kernel.ipengineapp.IEngineApp class method\), 502](#)
[__init__\(\) \(IPython.kernel.ipengineapp.IEngineAppConfigLoader class method\), 504](#)
[__init__\(\) \(IPython.kernel.mapper.IMapper class method\), 507](#)
[__init__\(\) \(IPython.kernel.mapper.IMultiEngineMapperFactory class method\), 509](#)
[__init__\(\) \(IPython.kernel.mapper.ITaskMapperFactory class method\), 511](#)
[__init__\(\) \(IPython.kernel.mapper.MultiEngineMapper class method\), 514](#)
[__init__\(\) \(IPython.kernel.mapper.SynchronousTaskMapper class method\), 514](#)
[__init__\(\) \(IPython.kernel.mapper.TaskMapper class method\), 515](#)
[__init__\(\) \(IPython.kernel.multiengine.IEngineMultiplexer class method\), 516](#)
[__init__\(\) \(IPython.kernel.multiengine.IFullMultiEngine class method\), 519](#)
[__init__\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine class method\), 521](#)
[__init__\(\) \(IPython.kernel.multiengine.IMultiEngine class method\), 523](#)
[__init__\(\) \(IPython.kernel.multiengine.IMultiEngineCoordinator class method\), 526](#)
[__init__\(\) \(IPython.kernel.multiengine.IMultiEngineExtras class method\), 528](#)
[__init__\(\) \(IPython.kernel.multiengine.ISynchronousEngineMultiplexer class method\), 530](#)
[__init__\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 533](#)
[__init__\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method\), 535](#)
[__init__\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method\), 538](#)
[__init__\(\) \(IPython.kernel.multiengine.MultiEngine class method\), 540](#)
[__init__\(\) \(IPython.kernel.multiengine.SynchronousMultiEngine class method\), 541](#)
[__init__\(\) \(IPython.kernel.multiengineclient.FullBlockingMultiEngine class method\), 543](#)
[__init__\(\) \(IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method\), 549](#)
[__init__\(\) \(IPython.kernel.multiengineclient.IPendingResult class method\), 551](#)
[__init__\(\) \(IPython.kernel.multiengineclient.InteractiveMultiEngineCoordinator class method\), 553](#)
[__init__\(\) \(IPython.kernel.multiengineclient.PendingResult class method\), 554](#)
[__init__\(\) \(IPython.kernel.multiengineclient.QueueStatusList class method\), 555](#)
[__init__\(\) \(IPython.kernel.multiengineclient.ResultList class method\), 555](#)
[__init__\(\) \(IPython.kernel.multienginefc.FCFullSynchronousMultiEngine class method\), 557](#)
[__init__\(\) \(IPython.kernel.multienginefc.FCSynchronousMultiEngine class method\), 559](#)
[__init__\(\) \(IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method\), 560](#)
[__init__\(\) \(IPython.kernel.newserialized.ISerialized class method\), 563](#)
[__init__\(\) \(IPython.kernel.newserialized.IUnSerialized class method\), 565](#)
[__init__\(\) \(IPython.kernel.newserialized.SerializeIt class method\), 568](#)
[__init__\(\) \(IPython.kernel.newserialized.Serialized class method\), 568](#)
[__init__\(\) \(IPython.kernel.newserialized.UnSerializeIt class method\), 568](#)
[__init__\(\) \(IPython.kernel.newserialized.UnSerialized class method\), 568](#)
[__init__\(\) \(IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method\), 569](#)

<code>__init__()</code> (IPython.kernel.parallelfunction.IParallelFunction method), 614	<code>__init__()</code> (IPython.lib.irunner.PythonRunner class method), 572
<code>__init__()</code> (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 574	<code>__init__()</code> (IPython.lib.irunner.RunnerFactory method), 616
<code>__init__()</code> (IPython.kernel.parallelfunction.ParallelFunction method), 617	<code>__init__()</code> (IPython.lib.irunner.SAGERunner method), 617
<code>__init__()</code> (IPython.kernel.pendingdeferred.PendingDeferredManager method), 578	<code>__init__()</code> (IPython.testing.globalipapp.ipnsdict method), 625
<code>__init__()</code> (IPython.kernel.pickleutil.CannedFunction method), 579	<code>__init__()</code> (IPython.testing.globalipapp.py_file_finder method), 626
<code>__init__()</code> (IPython.kernel.pickleutil.CannedObject method), 579	<code>__init__()</code> (IPython.testing.ipctest.IPTester method), 627
<code>__init__()</code> (IPython.kernel.twistedutil.DeferredList method), 581	<code>__init__()</code> (IPython.testing.ipunitest.Doc2UnitTester method), 629
<code>__init__()</code> (IPython.kernel.twistedutil.ReactorInThread method), 582	<code>__init__()</code> (IPython.testing.ipunitest.IPython2PythonConverter method), 629
<code>__init__()</code> (IPython.lib.backgroundjobs.BackgroundJobBase method), 585	<code>__init__()</code> (IPython.testing.mkdoctests.IndentOut method), 630
<code>__init__()</code> (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586	<code>__init__()</code> (IPython.testing.mkdoctests.RunnerFactory method), 631
<code>__init__()</code> (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587	<code>__init__()</code> (IPython.testing.plugin.show_refs.C method), 635
<code>__init__()</code> (IPython.lib.backgroundjobs.BackgroundJobManagement method), 588	<code>__init__()</code> (IPython.testing.tools.TempFileMixin method), 638
<code>__init__()</code> (IPython.lib.demo.ClearDemo method), 593	<code>__init__()</code> (IPython.utils.PyColorize.Parser method), 640
<code>__init__()</code> (IPython.lib.demo.ClearIPDemo method), 595	<code>__init__()</code> (IPython.utils.autoattr.OneTimeProperty method), 644
<code>__init__()</code> (IPython.lib.demo.ClearMixin method), 597	<code>__init__()</code> (IPython.utils.autoattr.ResetMixin method), 644
<code>__init__()</code> (IPython.lib.demo.Demo method), 597	<code>__init__()</code> (IPython.utils.coloransi.ColorScheme method), 646
<code>__init__()</code> (IPython.lib.demo.DemoError method), 599	<code>__init__()</code> (IPython.utils.coloransi.ColorSchemeTable method), 646
<code>__init__()</code> (IPython.lib.demo.IPythonDemo method), 599	<code>__init__()</code> (IPython.utils.growl.IPythonGrowlError method), 652
<code>__init__()</code> (IPython.lib.demo.IPythonLineDemo method), 601	<code>__init__()</code> (IPython.utils.growl.Notifier method), 653
<code>__init__()</code> (IPython.lib.demo.LineDemo method), 602	<code>__init__()</code> (IPython.utils.io.IOStream method), 654
<code>__init__()</code> (IPython.lib.inputhook.InputHookManager method), 606	<code>__init__()</code> (IPython.utils.io.IOTerm method), 654
<code>__init__()</code> (IPython.lib.inputhookwx.EventLoopRunner method), 609	<code>__init__()</code> (IPython.utils.io.NLprinter method), 655
<code>__init__()</code> (IPython.lib.inputhookwx.EventLoopTimer method), 609	<code>__init__()</code> (IPython.utils.io.Tee method), 655
<code>__init__()</code> (IPython.lib.irunner.IPythonRunner method), 614	<code>__init__()</code> (IPython.utils.ipstruct.Struct method), 657
<code>__init__()</code> (IPython.lib.irunner.InteractiveRunner method), 614	<code>__init__()</code> (IPython.utils.notification.NotificationCenter method), 662
	<code>__init__()</code> (IPython.utils.notification.NotificationError method), 662

method), 663

`__init__()` (IPython.utils.path.HomeDirError method), 664

`__init__()` (IPython.utils.pickleshare.PickleShareDB method), 667

`__init__()` (IPython.utils.pickleshare.PickleShareLink method), 668

`__init__()` (IPython.utils.process.FindCmdError method), 669

`__init__()` (IPython.utils.strdispatch.StrDispatch method), 670

`__init__()` (IPython.utils.syspathcontext.appended_to_syspath method), 672

`__init__()` (IPython.utils.syspathcontext.prepended_to_syspath method), 672

`__init__()` (IPython.utils.text.LSString method), 674

`__init__()` (IPython.utils.text.SList method), 679

`__init__()` (IPython.utils.wildcard.NameSpace method), 684

A

`abbrev_cwd()` (in module IPython.utils.process), 669

`abortCommand()` (IPython.kernel.engineservice.QueuedEngine method), 485

`AbortedPendingDeferredError` (class in IPython.kernel.error), 490

`activate()` (IPython.core.builtin_trap.BuiltinTrap method), 218

`activate()` (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 543

`activate()` (IPython.kernel.multiengineclient.InteractiveMultiEngineClient method), 553

`activate_matplotlib()` (in module IPython.lib.pylabtools), 619

`adapt_to_blocking_client()` (IPython.kernel.multienginefc.FCFullSynchronousClient method), 557

`add()` (IPython.core.history.ShadowHist method), 273

`add()` (IPython.core.hooks.CommandChainDispatcher method), 276

`add()` (IPython.kernel.core.util.InputList method), 457

`add_argument()` (IPython.config.loader.ArgumentParser method), 204

`add_argument_group()` (IPython.config.loader.ArgumentParser method), 204

`add_builtin()` (IPython.core.builtin_trap.BuiltinTrap method), 218

`add_callback()` (IPython.kernel.multiengineclient.PendingResult method), 554

`add_items()` (IPython.kernel.core.history.FrontEndHistory method), 438

`add_message()` (IPython.kernel.core.message_cache.IMessageCache method), 446

`add_message()` (IPython.kernel.core.message_cache.SimpleMessageCache method), 447

`add_mutually_exclusive_group()` (IPython.config.loader.ArgumentParser method), 204

`add_notifier()` (IPython.utils.notification.NotificationCenter method), 662

`add_re()` (IPython.utils.strdispatch.StrDispatch method), 670

`add_s()` (IPython.utils.strdispatch.StrDispatch method), 670

`add_scheme()` (IPython.utils.coloransi.ColorSchemeTable method), 646

`add_subparsers()` (IPython.config.loader.ArgumentParser method), 205

`add_to_message()` (IPython.kernel.core.display_trap.DisplayTrap method), 434

`add_to_message()` (IPython.kernel.core.output_trap.OutputTrap method), 447

`add_to_message()` (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), 452

`add_to_message()` (IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method), 453

`add_to_message()` (IPython.kernel.core.traceback_trap.TracebackTrap method), 455

`addBoth()` (IPython.kernel.twistedutil.DeferredList method), 581

`addCallback()` (IPython.kernel.twistedutil.DeferredList method), 581

`addCallbacks()` (IPython.kernel.twistedutil.DeferredList method), 581

`addErrback()` (IPython.kernel.twistedutil.DeferredList method), 581

`addIDToResult()` (IPython.kernel.engineservice.EngineService method), 469

`addIDToResult()` (IPython.kernel.engineservice.ThreadedEngineService method), 488

`AddPendingEvent()` (IPython.lib.inpthookwx.EventLoopTimer method), 609

`again()` (IPython.lib.demo.ClearDemo method), 594

- again() (IPython.lib.demo.ClearIPDemo method), 596
- again() (IPython.lib.demo.Demo method), 598
- again() (IPython.lib.demo.IPythonDemo method), 600
- again() (IPython.lib.demo.IPythonLineDemo method), 601
- again() (IPython.lib.demo.LineDemo method), 603
- alias_manager (IPython.core.interactiveshell.InteractiveShell attribute), 285
- alias_matches() (IPython.core.completer.IPCompleter method), 224
- AliasChecker (class in IPython.core.prefilter), 362
- AliasError (class in IPython.core.alias), 209
- aliases (IPython.core.alias.AliasManager attribute), 210
- AliasHandler (class in IPython.core.prefilter), 363
- AliasManager (class in IPython.core.alias), 210
- all() (IPython.core.history.ShadowHist method), 273
- all_belong() (in module IPython.utils.attic), 642
- all_completions() (IPython.core.completer.IPCompleter method), 224
- allow_new_attr() (IPython.utils.ipstruct.Struct method), 657
- append (IPython.kernel.core.util.InputList attribute), 457
- append (IPython.kernel.multiengineclient.QueueStatusList attribute), 555
- append (IPython.kernel.multiengineclient.ResultList attribute), 555
- append (IPython.utils.text.SList attribute), 679
- appended_to_syspath (class in IPython.utils.syspathcontext), 672
- Application (class in IPython.core.application), 213
- ApplicationError (class in IPython.core.application), 216
- apply_wrapper() (in module IPython.testing.decorators), 621
- arg_err() (IPython.core.interactiveshell.InteractiveShell method), 285
- arg_err() (IPython.core.magic.Magic method), 327
- arg_split() (in module IPython.utils.process), 669
- ArgParseConfigLoader (class in IPython.config.loader), 203
- args (IPython.config.loader.ConfigError attribute), 207
- args (IPython.config.loader.ConfigLoaderError attribute), 208
- args (IPython.core.alias.AliasError attribute), 210
- args (IPython.core.alias.InvalidAliasError attribute), 212
- args (IPython.core.application.ApplicationError attribute), 216
- args (IPython.core.error.IPythonCoreError attribute), 247
- args (IPython.core.error.TryNext attribute), 247
- args (IPython.core.error.UsageError attribute), 248
- args (IPython.core.interactiveshell.MultipleInstanceError attribute), 322
- args (IPython.core.interactiveshell.SpaceInInput attribute), 323
- args (IPython.core.prefilter.PrefilterError attribute), 384
- args (IPython.kernel.clientconnector.ClientConnectorError attribute), 417
- args (IPython.kernel.clientconnector.ClusterStateError attribute), 420
- args (IPython.kernel.core.error.ControllerCreationError attribute), 435
- args (IPython.kernel.core.error.ControllerError attribute), 435
- args (IPython.kernel.core.error.EngineCreationError attribute), 435
- args (IPython.kernel.core.error.EngineError attribute), 435
- args (IPython.kernel.core.error.IPythonError attribute), 436
- args (IPython.kernel.engineconnector.EngineConnectorError attribute), 460
- args (IPython.kernel.error.AbortedPendingDeferredError attribute), 491
- args (IPython.kernel.error.ClientError attribute), 491
- args (IPython.kernel.error.CompositeError attribute), 491
- args (IPython.kernel.error.ConnectionError attribute), 491
- args (IPython.kernel.error.FileTimeoutError attribute), 492
- args (IPython.kernel.error.IdInUse attribute), 492
- args (IPython.kernel.error.InvalidClientID attribute), 492
- args (IPython.kernel.error.InvalidDeferredID attribute), 492
- args (IPython.kernel.error.InvalidEngineID attribute), 492
- args (IPython.kernel.error.InvalidProperty attribute), 492

- 493
- args (IPython.kernel.error.KernelError attribute), 493
- args (IPython.kernel.error.MessageSizeError attribute), 493
- args (IPython.kernel.error.MissingBlockArgument attribute), 493
- args (IPython.kernel.error.NoEnginesRegistered attribute), 494
- args (IPython.kernel.error.NotAPendingResult attribute), 494
- args (IPython.kernel.error.NotDefined attribute), 494
- args (IPython.kernel.error.PBMessageSizeError attribute), 494
- args (IPython.kernel.error.ProtocolError attribute), 495
- args (IPython.kernel.error.QueueCleared attribute), 495
- args (IPython.kernel.error.ResultAlreadyRetrieved attribute), 495
- args (IPython.kernel.error.ResultNotCompleted attribute), 495
- args (IPython.kernel.error.SecurityError attribute), 495
- args (IPython.kernel.error.SerializationError attribute), 496
- args (IPython.kernel.error.StopLocalExecution attribute), 496
- args (IPython.kernel.error.TaskAborted attribute), 496
- args (IPython.kernel.error.TaskRejectError attribute), 497
- args (IPython.kernel.error.TaskTimeout attribute), 497
- args (IPython.kernel.error.UnpickleableException attribute), 497
- args (IPython.lib.demo.DemoError attribute), 599
- args (IPython.utils.growl.IPythonGrowlError attribute), 653
- args (IPython.utils.notification.NotificationError attribute), 663
- args (IPython.utils.path.HomeDirError attribute), 664
- args (IPython.utils.process.FindCmdError attribute), 669
- ArgumentParser (class in IPython.config.loader), 204
- argv (IPython.core.application.Application attribute), 213
- as_unittest() (in module IPython.testing.decorators), 622
- ask_yes_no() (in module IPython.utils.io), 655
- ask_yes_no() (IPython.core.interactiveshell.InteractiveShell method), 285
- AssignMagicTransformer (class in IPython.core.prefilter), 364
- AssignmentChecker (class in IPython.core.prefilter), 367
- AssignSystemTransformer (class in IPython.core.prefilter), 365
- AsyncClientConnector (class in IPython.kernel.clientconnector), 411
- AsyncCluster (class in IPython.kernel.clientconnector), 414
- atexit_operations() (IPython.core.interactiveshell.InteractiveShell method), 285
- attempt() (IPython.core.application.Application method), 213
- attempt() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
- attempt() (IPython.kernel.ipengineapp.IPEngineApp method), 502
- attr_matches() (IPython.core.completer.Completer method), 222
- attr_matches() (IPython.core.completer.IPCompleter method), 224
- auto_attr() (in module IPython.utils.autoattr), 644
- auto_rewrite() (IPython.core.prompts.Prompt1 method), 397
- auto_rewrite() (IPython.kernel.core.prompts.Prompt1 method), 450
- auto_rewrite_input() (IPython.core.interactiveshell.InteractiveShell method), 285
- auto_stop (IPython.kernel.clientconnector.Cluster attribute), 418
- autocall (IPython.core.interactiveshell.InteractiveShell attribute), 285
- AutocallChecker (class in IPython.core.prefilter), 371
- AutoFormattedTB (class in IPython.core.ultratb), 401
- AutoHandler (class in IPython.core.prefilter), 368
- autoindent (IPython.core.interactiveshell.InteractiveShell attribute), 285
- automagic (IPython.core.interactiveshell.InteractiveShell

- attribute), 285
- AutoMagicChecker (class in IPython.core.prefilter), 369
- autosave_if_due() (IPython.core.history.HistoryManager method), 271
- B**
- back() (IPython.lib.demos.ClearDemo method), 594
- back() (IPython.lib.demos.ClearIPDemo method), 596
- back() (IPython.lib.demos.Demo method), 598
- back() (IPython.lib.demos.IPythonDemo method), 600
- back() (IPython.lib.demos.IPythonLineDemo method), 601
- back() (IPython.lib.demos.LineDemo method), 603
- BackgroundJobBase (class in IPython.lib.backgroundjobs), 585
- BackgroundJobExpr (class in IPython.lib.backgroundjobs), 586
- BackgroundJobFunc (class in IPython.lib.backgroundjobs), 587
- BackgroundJobManager (class in IPython.lib.backgroundjobs), 587
- barrier() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 543
- BaseAppConfigLoader (class in IPython.core.application), 216
- BaseFormatter (class in IPython.core.formatters), 251
- BasePrompt (class in IPython.core.prompts), 396
- BasePrompt (class in IPython.kernel.core.prompts), 449
- BdbQuit_excepthook() (in module IPython.core.debugger), 237
- BdbQuit_IPython_excepthook() (in module IPython.core.debugger), 237
- belong() (in module IPython.utils.attic), 642
- benchmark() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 544
- Bind() (IPython.lib.inpthookwx.EventLoopTimer method), 609
- bp_commands() (IPython.core.debugger.Pdb method), 229
- break_anywhere() (IPython.core.debugger.Pdb method), 229
- break_here() (IPython.core.debugger.Pdb method), 229
- builtin_trap (IPython.core.interactiveshell.InteractiveShell attribute), 286
- BuiltinTrap (class in IPython.core.builtin_trap), 218
- Bunch (class in IPython.core.completer), 222
- Bunch (class in IPython.core.interactiveshell), 284
- Bunch (class in IPython.core.magic), 326
- Bunch (class in IPython.kernel.core.util), 456
- C**
- C (class in IPython.testing.plugin.show_refs), 635
- cache_main_mod() (IPython.core.interactiveshell.InteractiveShell method), 286
- cache_size (IPython.core.interactiveshell.InteractiveShell attribute), 286
- CachedOutput (class in IPython.kernel.core.prompts), 449
- CachingCompiler (class in IPython.core.compilerop), 220
- call() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586
- call() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587
- call_alias() (IPython.core.alias.AliasManager method), 210
- call_connect() (IPython.kernel.ipengineapp.IPEngineApp method), 502
- call_pdb (IPython.core.interactiveshell.InteractiveShell attribute), 286
- call_tip() (in module IPython.core.oinspect), 352
- callback() (IPython.kernel.twistedutil.DeferredList method), 581
- callRemote() (IPython.kernel.enginefc.EngineFromReference method), 461
- can() (in module IPython.kernel.pickleutil), 580
- canDict() (in module IPython.kernel.pickleutil), 580
- CannedFunction (class in IPython.kernel.pickleutil), 580
- CannedObject (class in IPython.kernel.pickleutil), 579
- canonic() (IPython.core.debugger.Pdb method), 229
- canSequence() (in module IPython.kernel.pickleutil), 580
- capitalize (IPython.utils.text.LSString attribute), 674
- catcher() (in module IPython.kernel.util), 584
- cd_completer() (in module IPython.core.completerlib), 226

center (IPython.utils.text.LSString attribute), 674

chainDeferred() (IPython.kernel.twistedutil.DeferredList class method), 581

changed() (IPython.kernel.clientinterfaces.IBlockingClientAdapter class method), 420

changed() (IPython.kernel.clientinterfaces.IFCCClientInterface class method), 423

changed() (IPython.kernel.controllerservice.IControllerBase class method), 427

changed() (IPython.kernel.controllerservice.IControllerManager class method), 430

changed() (IPython.kernel.enginefc.IFCCControllerBase class method), 463

changed() (IPython.kernel.enginefc.IFCEngine class method), 466

changed() (IPython.kernel.engineservice.IEngineBase class method), 471

changed() (IPython.kernel.engineservice.IEngineCore class method), 473

changed() (IPython.kernel.engineservice.IEngineProperties class method), 475

changed() (IPython.kernel.engineservice.IEngineQueue class method), 478

changed() (IPython.kernel.engineservice.IEngineSerialized class method), 480

changed() (IPython.kernel.engineservice.IEngineThread class method), 482

changed() (IPython.kernel.mapper.IMapper class method), 507

changed() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 509

changed() (IPython.kernel.mapper.ITaskMapperFactory class method), 511

changed() (IPython.kernel.multiengine.IEngineMultiple class method), 517

changed() (IPython.kernel.multiengine.IFullMultiEngine class method), 519

changed() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 521

changed() (IPython.kernel.multiengine.IMultiEngine class method), 523

changed() (IPython.kernel.multiengine.IMultiEngineController class method), 526

changed() (IPython.kernel.multiengine.IMultiEngineExtension class method), 528

changed() (IPython.kernel.multiengine.ISynchronousEngine class method), 530

changed() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 533

changed() (IPython.kernel.multiengine.ISynchronousMultiEngineController class method), 535

changed() (IPython.kernel.multiengine.ISynchronousMultiEngineExtension class method), 538

changed() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), 549

changed() (IPython.kernel.multiengineclient.IPendingResult class method), 551

changed() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 560

changed() (IPython.kernel.newserialized.ISerialized class method), 563

changed() (IPython.kernel.newserialized.IUnSerialized class method), 565

changed() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 569

changed() (IPython.kernel.parallelfunction.IParallelFunction class method), 572

changed() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 574

check() (IPython.core.prefilter.AliasChecker method), 362

check() (IPython.core.prefilter.AssignmentChecker method), 367

check() (IPython.core.prefilter.AutocallChecker method), 371

check() (IPython.core.prefilter.AutoMagicChecker method), 369

check() (IPython.core.prefilter.EmacsChecker method), 372

check() (IPython.core.prefilter.EscCharsChecker method), 375

check() (IPython.core.prefilter.IPyAutocallChecker method), 377

check() (IPython.core.prefilter.MultiLineMagicChecker method), 382

check() (IPython.core.prefilter.PrefilterChecker method), 383

check() (IPython.core.prefilter.PythonOpsChecker method), 392

check() (IPython.core.prefilter.ShellEscapeChecker method), 393

clear_cache() (IPython.core.compilerop.CachingCompiler method), 220

clear_multipliers() (IPython.core.displayhook.DisplayHook method), 241

[check_stdin\(\)](#) (IPython.lib.inputhookwx.EventLoopRunner method), [505](#)
[checkers](#) (IPython.core.prefilter.PrefilterManager attribute), [386](#)
[checkline\(\)](#) (IPython.core.debugger.Pdb method), [230](#)
[checkMessageSize\(\)](#) (in module IPython.kernel.pbutil), [577](#)
[checkReturnForFailure\(\)](#) (IPython.kernel.enginefc.EngineFromReference method), [461](#)
[chop\(\)](#) (in module IPython.utils.data), [648](#)
[ClassName](#) (IPython.lib.inputhookwx.EventLoopTimer attribute), [610](#)
[cleanup\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [286](#)
[clear](#) (IPython.config.loader.Config attribute), [206](#)
[clear](#) (IPython.kernel.core.util.Bunch attribute), [456](#)
[clear](#) (IPython.utils.coloransi.ColorSchemeTable attribute), [646](#)
[clear](#) (IPython.utils.ipstruct.Struct attribute), [657](#)
[clear\(\)](#) (IPython.config.loader.ArgParseConfigLoader method), [204](#)
[clear\(\)](#) (IPython.config.loader.CommandLineConfigLoader method), [205](#)
[clear\(\)](#) (IPython.config.loader.ConfigLoader method), [207](#)
[clear\(\)](#) (IPython.config.loader.FileConfigLoader method), [208](#)
[clear\(\)](#) (IPython.config.loader.PyFileConfigLoader method), [209](#)
[clear\(\)](#) (IPython.core.application.BaseAppConfigLoader method), [216](#)
[clear\(\)](#) (IPython.kernel.core.display_trap.DisplayTrap method), [434](#)
[clear\(\)](#) (IPython.kernel.core.output_trap.OutputTrap method), [448](#)
[clear\(\)](#) (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), [452](#)
[clear\(\)](#) (IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method), [454](#)
[clear\(\)](#) (IPython.kernel.core.traceback_trap.TracebackTrap method), [455](#)
[clear\(\)](#) (IPython.kernel.engineservice.StrictDict method), [487](#)
[clear\(\)](#) (IPython.kernel.ipclusterapp.IPClusterAppConfigLoader method), [501](#)
[clear\(\)](#) (IPython.kernel.ipengineapp.IPEngineAppConfigLoader method), [540](#)
[clear\(\)](#) (IPython.testing.globalipapp.ipnsdict method), [625](#)
[clear\(\)](#) (IPython.utils.pickleshare.PickleShareDB method), [667](#)
[clear_aliases\(\)](#) (IPython.core.alias.AliasManager method), [210](#)
[clear_all_breaks\(\)](#) (IPython.core.debugger.Pdb method), [230](#)
[clear_all_file_breaks\(\)](#) (IPython.core.debugger.Pdb method), [230](#)
[clear_app_refs\(\)](#) (IPython.lib.inputhook.InputHookManager method), [606](#)
[clear_bpbynumber\(\)](#) (IPython.core.debugger.Pdb method), [230](#)
[clear_break\(\)](#) (IPython.core.debugger.Pdb method), [230](#)
[clear_err_state\(\)](#) (IPython.core.ultratb.SyntaxTB method), [407](#)
[clear_inputhook\(\)](#) (IPython.lib.inputhook.InputHookManager method), [606](#)
[clear_main_mod_cache\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [286](#)
[clear_payload\(\)](#) (IPython.core.payload.PayloadManager method), [355](#)
[clear_pending_deferreds\(\)](#) (IPython.kernel.multiengine.SynchronousMultiEngine method), [541](#)
[clear_pending_deferreds\(\)](#) (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), [557](#)
[clear_pending_deferreds\(\)](#) (IPython.kernel.pendingdeferred.PendingDeferredManager method), [578](#)
[clear_pending_results\(\)](#) (IPython.kernel.multiengineclient.FullBlockingMultiEngine method), [544](#)
[clear_properties\(\)](#) (IPython.kernel.enginefc.EngineFromReference method), [461](#)
[clear_properties\(\)](#) (IPython.kernel.engineservice.EngineService method), [469](#)
[clear_properties\(\)](#) (IPython.kernel.engineservice.QueuedEngine method), [485](#)
[clear_properties\(\)](#) (IPython.kernel.engineservice.ThreadedEngineService method), [488](#)
[clear_properties\(\)](#) (IPython.kernel.multiengine.MultiEngine method), [540](#)

clear_properties() (IPython.kernel.multiengine.SynchronousMultiEngine method), 541

clear_properties() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 544

clear_properties() (IPython.kernel.multienginefc.FCFullSyncExceptionMultiEngineClient method), 557

clear_queue() (IPython.kernel.engineservice.QueuedEngineInfo attribute), 485

clear_queue() (IPython.kernel.multiengine.MultiEngine color_toggle() (IPython.core.ultratb.AutoFormattedTB method), 540

clear_queue() (IPython.kernel.multiengine.SynchronousMultiEngine color_toggle() (IPython.core.ultratb.ColorTB method), 541

clear_queue() (IPython.kernel.multiengineclient.FullBlockingMultiEngine color_toggle() (IPython.core.ultratb.FormattedTB method), 544

clear_queue() (IPython.kernel.multienginefc.FCFullSyncExceptionMultiEngine color_toggle() (IPython.core.ultratb.ListTB method), 557

ClearDemo (class in IPython.lib.demo), 593

ClearIPDemo (class in IPython.lib.demo), 595

ClearMixin (class in IPython.lib.demo), 597

ClientConnector (class in IPython.kernel.clientconnector), 415

ClientConnectorError (class in IPython.kernel.clientconnector), 417

ClientError (class in IPython.kernel.error), 491

clipboard_get() (in module IPython.core.hooks), 276

close() (IPython.core.oinspect.myStringIO method), 351

close() (IPython.kernel.core.file_like.FileLike method), 437

close() (IPython.lib.irunner.InteractiveRunner method), 615

close() (IPython.lib.irunner.IPythonRunner method), 614

close() (IPython.lib.irunner.PythonRunner method), 616

close() (IPython.lib.irunner.SAGERunner method), 617

close() (IPython.testing.mkdoctests.IndentOut method), 631

close() (IPython.utils.io.IOStream method), 654

close() (IPython.utils.io.Tee method), 655

close_log() (IPython.core.logger.Logger method), 324

Cluster (class in IPython.kernel.clientconnector), 417

ClusterStateError (class in IPython.kernel.clientconnector), 420

cmdloop() (IPython.core.debugger.Pdb method), 217

code_ctor() (in module IPython.kernel.codeutil), 425

compile_ipython_code() (in module IPython.core.compilerop), 220

compile_ipython_code() (in module IPython.kernel.error), 497

color_toggle() (IPython.core.ultratb.AutoFormattedTB method), 401

color_toggle() (IPython.core.ultratb.ColorTB method), 402

color_toggle() (IPython.core.ultratb.FormattedTB method), 404

color_toggle() (IPython.core.ultratb.ListTB method), 406

color_toggle() (IPython.core.ultratb.SyntaxTB method), 407

color_toggle() (IPython.core.ultratb.TBTools method), 408

color_toggle() (IPython.core.ultratb.VerboseTB method), 409

colors (IPython.core.interactiveshell.InteractiveShell attribute), 287

ColorScheme (class in IPython.utils.coloransi), 646

ColorSchemeTable (class in IPython.utils.coloransi), 646

ColorTB (class in IPython.core.ultratb), 402

columnize() (IPython.core.debugger.Pdb method), 230

Command (class in IPython.kernel.engineservice), 469

command_line_config (IPython.core.application.Application attribute), 213

command_line_loader (IPython.core.application.Application attribute), 213

command_line_loader (IPython.kernel.ipclusterapp.IPClusterApp attribute), 498

command_line_loader (IPython.kernel.ipengineapp.IPEngineApp attribute), 502

CommandChainDispatcher (class in IPython.core.hooks), 276

CommandLineConfigLoader (class in IPython.config.loader), 205

compiler_flags (IPython.core.compilerop.CachingCompiler attribute), 220	config (IPython.core.formatters.BaseFormatter attribute), 252
complete() (IPython.core.completer.Completer method), 222	config (IPython.core.formatters.DisplayFormatter attribute), 254
complete() (IPython.core.completer.IPCompleter method), 224	config (IPython.core.formatters.HTMLFormatter attribute), 256
complete() (IPython.core.debugger.Pdb method), 230	config (IPython.core.formatters.JSONFormatter attribute), 259
complete() (IPython.core.interactiveshell.InteractiveShell method), 287	config (IPython.core.formatters.LatexFormatter attribute), 261
complete() (IPython.kernel.core.interpreter.Interpreter method), 440	config (IPython.core.formatters.PlainTextFormatter attribute), 266
complete_help() (IPython.core.debugger.Pdb method), 230	config (IPython.core.formatters.PNGFormatter attribute), 263
complete_object() (in module IPython.utils.generics), 652	config (IPython.core.formatters.SVGFormatter attribute), 268
completedefault() (IPython.core.debugger.Pdb method), 230	config (IPython.core.interactiveshell.InteractiveShell attribute), 287
completenames() (IPython.core.debugger.Pdb method), 230	config (IPython.core.payload.PayloadManager attribute), 355
Completer (class in IPython.core.completer), 222	config (IPython.core.plugin.Plugin attribute), 358
CompletionSplitter (class in IPython.core.completer), 223	config (IPython.core.plugin.PluginManager attribute), 359
CompositeError (class in IPython.kernel.error), 491	config (IPython.core.prefilter.AliasChecker attribute), 362
compress_dhist() (in module IPython.core.magic), 348	config (IPython.core.prefilter.AliasHandler attribute), 363
compress_user() (in module IPython.core.completer), 225	config (IPython.core.prefilter.AssignMagicTransformer attribute), 364
compute_format_data() (IPython.core.displayhook.DisplayHook method), 241	config (IPython.core.prefilter.AssignmentChecker attribute), 367
concatenate() (IPython.kernel.map.Map method), 506	config (IPython.core.prefilter.AssignSystemTransformer attribute), 366
concatenate() (IPython.kernel.map.RoundRobinMap method), 506	config (IPython.core.prefilter.AutocallChecker attribute), 371
Config (class in IPython.config.loader), 206	config (IPython.core.prefilter.AutoHandler attribute), 368
config (IPython.core.alias.AliasManager attribute), 210	config (IPython.core.prefilter.AutoMagicChecker attribute), 369
config (IPython.core.builtin_trap.BuiltinTrap attribute), 218	config (IPython.core.prefilter.EmacsChecker attribute), 372
config (IPython.core.display_trap.DisplayTrap attribute), 239	config (IPython.core.prefilter.EmacsHandler attribute), 373
config (IPython.core.displayhook.DisplayHook attribute), 241	config (IPython.core.prefilter.EscCharsChecker attribute), 375
config (IPython.core.displaypub.DisplayPublisher attribute), 244	config (IPython.core.prefilter.HelpHandler attribute), 376
config (IPython.core.extensions.ExtensionManager attribute), 249	config (IPython.core.prefilter.IPyAutocallChecker attribute), 377

attribute), 377

config (IPython.core.prefilter.IPyPromptTransformer attribute), 379

config (IPython.core.prefilter.MagicHandler attribute), 380

config (IPython.core.prefilter.MultiLineMagicChecker attribute), 382

config (IPython.core.prefilter.PrefilterChecker attribute), 383

config (IPython.core.prefilter.PrefilterHandler attribute), 385

config (IPython.core.prefilter.PrefilterManager attribute), 387

config (IPython.core.prefilter.PrefilterTransformer attribute), 389

config (IPython.core.prefilter.PyPromptTransformer attribute), 390

config (IPython.core.prefilter.PythonOpsChecker attribute), 392

config (IPython.core.prefilter.ShellEscapeChecker attribute), 393

config (IPython.core.prefilter.ShellEscapeHandler attribute), 394

config_file_name (IPython.core.application.Application attribute), 213

ConfigError (class in IPython.config.loader), 207

ConfigLoader (class in IPython.config.loader), 207

ConfigLoaderError (class in IPython.config.loader), 208

Connect() (IPython.lib.inpthookwx.EventLoopTimer method), 610

connect_to_controller() (IPython.kernel.engineconnector.EngineConnector method), 460

ConnectionError (class in IPython.kernel.error), 491

construct() (IPython.core.application.Application method), 213

construct() (IPython.kernel.ipclusterapp.IPClusterApp method), 498

construct() (IPython.kernel.ipengineapp.IPEngineApp method), 502

context() (IPython.core.ultratb.AutoFormattedTB method), 401

context() (IPython.core.ultratb.ColorTB method), 402

context() (IPython.core.ultratb.FormattedTB method), 404

ControllerAdapterBase (class in IPython.kernel.controllerservice), 426

ControllerCreationError (class in IPython.kernel.core.error), 434

ControllerError (class in IPython.kernel.core.error), 435

ControllerService (class in IPython.kernel.controllerservice), 427

copy (IPython.kernel.core.util.Bunch attribute), 456

copy (IPython.kernel.engineservice.StrictDict attribute), 487

copy (IPython.testing.globalipapp.ipnsdict attribute), 625

copy() (IPython.config.loader.Config method), 206

copy() (IPython.utils.coloransi.ColorScheme method), 646

copy() (IPython.utils.coloransi.ColorSchemeTable method), 646

copy() (IPython.utils.ipstruct.Struct method), 657

count (IPython.kernel.core.util.InputList attribute), 457

count (IPython.kernel.multiengineclient.QueueStatusList attribute), 555

count (IPython.kernel.multiengineclient.ResultList attribute), 556

count (IPython.utils.text.LSString attribute), 674

count (IPython.utils.text.SList attribute), 679

count_failures() (in IPython.testing.ipunittest), 629

crash_handler_class (IPython.core.application.Application attribute), 213

crash_handler_class (IPython.kernel.ipclusterapp.IPClusterApp attribute), 498

crash_handler_class (IPython.kernel.ipengineapp.IPEngineApp attribute), 502

CrashHandler (class in IPython.core.crashhandler), 228

create_command_line_config() (IPython.core.application.Application method), 213

create_command_line_config() (IPython.kernel.ipclusterapp.IPClusterApp method), 498

create_command_line_config() (IPython.kernel.ipengineapp.IPEngineApp method), 502

- `create_crash_handler()`
(`IPython.core.application.Application` method), 213
 - `create_crash_handler()`
(`IPython.kernel.ipclusterapp.IPClusterApp` method), 498
 - `create_crash_handler()`
(`IPython.kernel.ipengineapp.IPEngineApp` method), 502
 - `create_default_config()`
(`IPython.core.application.Application` method), 213
 - `create_default_config()`
(`IPython.kernel.ipclusterapp.IPClusterApp` method), 498
 - `create_default_config()`
(`IPython.kernel.ipengineapp.IPEngineApp` method), 502
 - `create_typestr2type_dicts()` (in module `IPython.utils.wildcard`), 685
 - `current_gui()` (`IPython.lib.inputhook.InputHookManager` method), 606
 - `curry()` (in module `IPython.kernel.util`), 584
 - `cwd_filt()` (`IPython.core.prompts.BasePrompt` method), 396
 - `cwd_filt()` (`IPython.core.prompts.Prompt1` method), 397
 - `cwd_filt()` (`IPython.core.prompts.Prompt2` method), 397
 - `cwd_filt()` (`IPython.core.prompts.PromptOut` method), 398
 - `cwd_filt()` (`IPython.kernel.core.prompts.BasePrompt` method), 449
 - `cwd_filt()` (`IPython.kernel.core.prompts.Prompt1` method), 450
 - `cwd_filt()` (`IPython.kernel.core.prompts.Prompt2` method), 450
 - `cwd_filt()` (`IPython.kernel.core.prompts.PromptOut` method), 451
 - `cwd_filt2()` (`IPython.core.prompts.BasePrompt` method), 396
 - `cwd_filt2()` (`IPython.core.prompts.Prompt1` method), 397
 - `cwd_filt2()` (`IPython.core.prompts.Prompt2` method), 398
 - `cwd_filt2()` (`IPython.core.prompts.PromptOut` method), 398
 - `cwd_filt2()` (`IPython.kernel.core.prompts.BasePrompt` method), 449
 - `cwd_filt2()` (`IPython.kernel.core.prompts.Prompt1` method), 450
 - `cwd_filt2()` (`IPython.kernel.core.prompts.Prompt2` method), 451
 - `cwd_filt2()` (`IPython.kernel.core.prompts.PromptOut` method), 451
- ## D
- `daemon` (`IPython.kernel.twistedutil.ReactorInThread` attribute), 582
 - `daemon` (`IPython.lib.backgroundjobs.BackgroundJobBase` attribute), 585
 - `daemon` (`IPython.lib.backgroundjobs.BackgroundJobExpr` attribute), 586
 - `daemon` (`IPython.lib.backgroundjobs.BackgroundJobFunc` attribute), 587
 - `db` (`IPython.core.interactiveshell.InteractiveShell` attribute), 287
 - `deactivate()` (`IPython.core.builtin_trap.BuiltinTrap` method), 218
 - `debug` (`IPython.core.interactiveshell.InteractiveShell` attribute), 287
 - `debugger()` (`IPython.core.interactiveshell.InteractiveShell` method), 288
 - `debugger()` (`IPython.core.ultratb.AutoFormattedTB` method), 401
 - `debugger()` (`IPython.core.ultratb.ColorTB` method), 402
 - `debugger()` (`IPython.core.ultratb.FormattedTB` method), 404
 - `debugger()` (`IPython.core.ultratb.VerboseTB` method), 409
 - `debugx()` (in module `IPython.utils.frame`), 651
 - `decode` (`IPython.utils.text.LSString` attribute), 674
 - `decorate_fn_with_doc()` (in module `IPython.core.debugger`), 237
 - `deep_import_hook()` (in module `IPython.lib.deepreload`), 590
 - `deep_reload` (`IPython.core.interactiveshell.InteractiveShell` attribute), 288
 - `deep_reload_hook()` (in module `IPython.lib.deepreload`), 590
 - `default()` (`IPython.core.debugger.Pdb` method), 230
 - `default_aliases` (`IPython.core.alias.AliasManager` attribute), 210
 - `default_aliases()` (in module `IPython.core.alias`), 212
 - `default_argv()` (in module `IPython.testing.tools`), 638

default_config (IPython.core.application.Application attribute), 214

default_config() (in module IPython.testing.tools), 638

default_config_file_name (IPython.core.application.Application attribute), 214

default_display_formatters() (in module IPython.kernel.core.interpreter), 444

default_option() (IPython.core.interactiveshell.InteractiveShell method), 288

default_option() (IPython.core.magic.Magic method), 327

default_traceback_formatters() (in module IPython.kernel.core.interpreter), 444

defaultFile() (IPython.core.debugger.Pdb method), 230

deferred() (IPython.kernel.clientinterfaces.IBlockingClientInterface class method), 420

deferred() (IPython.kernel.clientinterfaces.IFCClientInterface class method), 423

deferred() (IPython.kernel.controllerservice.IControllerBased class method), 428

deferred() (IPython.kernel.controllerservice.IControllerBased class method), 430

deferred() (IPython.kernel.enginefc.IFCControllerBased class method), 463

deferred() (IPython.kernel.enginefc.IFCEngine class method), 466

deferred() (IPython.kernel.engineservice.IEngineBase class method), 471

deferred() (IPython.kernel.engineservice.IEngineCore class method), 473

deferred() (IPython.kernel.engineservice.IEngineProperties class method), 475

deferred() (IPython.kernel.engineservice.IEngineQueue class method), 478

deferred() (IPython.kernel.engineservice.IEngineSerialized class method), 480

deferred() (IPython.kernel.engineservice.IEngineThread class method), 483

deferred() (IPython.kernel.mapper.IMapper class method), 507

deferred() (IPython.kernel.mapper.IMultiEngineMapper class method), 509

deferred() (IPython.kernel.mapper.ITaskMapperFactory class method), 511

deferred() (IPython.kernel.multiengine.IEngineMultiple class method), 517

deferred() (IPython.kernel.multiengine.IFullMultiEngine class method), 519

deferred() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 521

deferred() (IPython.kernel.multiengine.IMultiEngine class method), 524

deferred() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 526

deferred() (IPython.kernel.multiengine.IMultiEngineExtras class method), 528

deferred() (IPython.kernel.multiengine.ISynchronousEngineMultiple class method), 530

deferred() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 533

deferred() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 535

deferred() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method), 538

deferred() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), 549

deferred() (IPython.kernel.multiengineclient.IPendingResult class method), 551

deferred() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 560

deferred() (IPython.kernel.newserialized.ISerialized class method), 563

deferred() (IPython.kernel.newserialized.IUnSerialized class method), 565

deferred() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 569

deferred() (IPython.kernel.parallelfunction.IParallelFunction class method), 572

deferred() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 574

deferred_printers (IPython.core.formatters.BaseFormatter attribute), 252

deferred_printers (IPython.core.formatters.HTMLFormatter attribute), 257

deferred_printers (IPython.core.formatters.JSONFormatter attribute), 259

deferred_printers (IPython.core.formatters.LatexFormatter attribute), 261

deferred_printers (IPython.core.formatters.PlainTextFormatter attribute), 266

deferred_printers (IPython.core.formatters.PNGFormatter attribute), 263

deferred_printers (IPython.core.formatters.SVGFormatter attribute), 263

- attribute), [268](#)
- DeferredList (class in IPython.kernel.twistedutil), [580](#)
- define_alias() (IPython.core.alias.AliasManager method), [210](#)
- define_macro() (IPython.core.interactiveshell.InteractiveShell method), [288](#)
- define_magic() (IPython.core.interactiveshell.InteractiveShell method), [288](#)
- del_properties() (IPython.kernel.enginefc.EngineFromReference method), [461](#)
- del_properties() (IPython.kernel.engineservice.EngineService method), [469](#)
- del_properties() (IPython.kernel.engineservice.QueuedEngine method), [485](#)
- del_properties() (IPython.kernel.engineservice.ThreadedEngine method), [488](#)
- del_properties() (IPython.kernel.multiengine.MultiEngine method), [540](#)
- del_properties() (IPython.kernel.multiengine.SynchronousMultiEngine method), [541](#)
- del_properties() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), [544](#)
- del_properties() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient method), [557](#)
- delete_pending_deferred() (IPython.kernel.multiengine.SynchronousMultiEngine class method), [511](#)
- delete_pending_deferred() (IPython.kernel.pendingdeferred.PendingDeferred method), [578](#)
- DeletePendingEvents() (IPython.lib.inputhookwx.EventLoopTimer method), [610](#)
- Demo (class in IPython.lib.demo), [597](#)
- DemoError (class in IPython.lib.demo), [599](#)
- Destroy() (IPython.lib.inputhookwx.EventLoopTimer method), [610](#)
- determine_parent() (in module IPython.lib.deepreload), [590](#)
- dgrep() (in module IPython.utils.text), [681](#)
- dhook_wrap() (in module IPython.utils.doctestreload), [650](#)
- dict() (IPython.utils.ipstruct.Struct method), [658](#)
- dir2() (in module IPython.utils.dir2), [649](#)
- direct() (IPython.kernel.clientinterfaces.IBlockingClientAdapter class method), [420](#)
- direct() (IPython.kernel.clientinterfaces.IFCCClientInterface class method), [423](#)
- direct() (IPython.kernel.controllerservice.IControllerBase class method), [428](#)
- direct() (IPython.kernel.controllerservice.IControllerCore class method), [430](#)
- direct() (IPython.kernel.enginefc.IFCCControllerBase class method), [463](#)
- direct() (IPython.kernel.enginefc.IFCEngine class method), [466](#)
- direct() (IPython.kernel.engineservice.IEngineBase class method), [471](#)
- direct() (IPython.kernel.engineservice.IEngineCore class method), [473](#)
- direct() (IPython.kernel.engineservice.IEngineProperties class method), [476](#)
- direct() (IPython.kernel.engineservice.IEngineQueued class method), [478](#)
- direct() (IPython.kernel.engineservice.IEngineSerialized class method), [480](#)
- direct() (IPython.kernel.engineservice.IEngineThreaded class method), [483](#)
- direct() (IPython.kernel.multiengineclient.IMapper class method), [507](#)
- direct() (IPython.kernel.multiengineclient.IMultiEngineMapperFactory class method), [509](#)
- direct() (IPython.kernel.mapper.ITaskMapperFactory class method), [511](#)
- direct() (IPython.kernel.multiengine.IEngineMultiplexer class method), [517](#)
- direct() (IPython.kernel.multiengine.IFullMultiEngine class method), [519](#)
- direct() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), [521](#)
- direct() (IPython.kernel.multiengine.IMultiEngine class method), [524](#)
- direct() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), [526](#)
- direct() (IPython.kernel.multiengine.IMultiEngineExtras class method), [528](#)
- direct() (IPython.kernel.multiengine.ISynchronousEngineMultiplexer class method), [531](#)
- direct() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), [533](#)
- direct() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), [535](#)
- direct() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method), [538](#)
- direct() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), [540](#)

- class method), 549
- direct() (IPython.kernel.multiengineclient.IPendingResult class method), 551
- direct() (IPython.kernel.multienginefc.IFCSynchronousMultiEngineAttribute), 288
- class method), 560
- direct() (IPython.kernel.newserialized.ISerialized class method), 563
- direct() (IPython.kernel.newserialized.IUnSerialized class method), 566
- direct() (IPython.kernel.parallelfunction.IMultiEngineDisplayDecorator class method), 569
- direct() (IPython.kernel.parallelfunction.IParallelFunction class method), 572
- direct() (IPython.kernel.parallelfunction.ITaskParallelDecorator attribute), 288
- class method), 574
- disable_gtk() (IPython.lib.inputhook.InputHookManager method), 606
- disable_qt4() (IPython.lib.inputhook.InputHookManager method), 606
- disable_tk() (IPython.lib.inputhook.InputHookManager method), 607
- disable_wx() (IPython.lib.inputhook.InputHookManager method), 607
- Disconnect() (IPython.lib.inputhookwx.EventLoopTimer method), 610
- disownServiceParent() (IPython.kernel.controllerservice.ControllerService method), 427
- disownServiceParent() (IPython.kernel.engineservice.EngineService method), 470
- disownServiceParent() (IPython.kernel.engineservice.ThreadedEngineService method), 488
- dispatch() (IPython.utils.strdispatch.StrDispatch method), 670
- dispatch_call() (IPython.core.debugger.Pdb method), 230
- dispatch_custom_completer() (IPython.core.completer.IPCompleter method), 224
- dispatch_exception() (IPython.core.debugger.Pdb method), 230
- dispatch_line() (IPython.core.debugger.Pdb method), 230
- dispatch_return() (IPython.core.debugger.Pdb method), 230
- display() (in module IPython.core.display), 237
- display() (IPython.kernel.core.prompts.CachedOutput method), 449
- display_formatter (IPython.core.interactiveshell.InteractiveShell attribute), 288
- display_html() (in module IPython.core.display), 238
- display_json() (in module IPython.core.display), 238
- display_latex() (in module IPython.core.display), 238
- display_png() (in module IPython.core.display), 238
- display_pretty() (in module IPython.core.display), 238
- display_pub_class (IPython.core.interactiveshell.InteractiveShell attribute), 288
- display_svg() (in module IPython.core.display), 238
- display_trap (IPython.core.interactiveshell.InteractiveShell attribute), 288
- DisplayFormatter (class in IPython.core.formatters), 254
- DisplayHook (class in IPython.core.displayhook), 241
- displayhook_class (IPython.core.interactiveshell.InteractiveShell attribute), 288
- DisplayPublisher (class in IPython.core.displaypub), 243
- DisplayTrap (class in IPython.core.display_trap), 239
- DisplayTrap (class in IPython.kernel.core.display_trap), 433
- do_a() (IPython.core.debugger.Pdb method), 230
- do_alias() (IPython.core.debugger.Pdb method), 230
- do_args() (IPython.core.debugger.Pdb method), 231
- do_break() (IPython.core.debugger.Pdb method), 231
- do_bt() (IPython.core.debugger.Pdb method), 231
- do_c() (IPython.core.debugger.Pdb method), 231
- do_cl() (IPython.core.debugger.Pdb method), 231
- do_clear() (IPython.core.debugger.Pdb method), 231
- do_commands() (IPython.core.debugger.Pdb method), 231
- do_condition() (IPython.core.debugger.Pdb method), 231
- do_cont() (IPython.core.debugger.Pdb method), 231
- do_continue() (IPython.core.debugger.Pdb method), 231
- do_d() (IPython.core.debugger.Pdb method), 231
- do_debug() (IPython.core.debugger.Pdb method), 231

[231](#)
`do_disable()` (IPython.core.debugger.Pdb method), [231](#)
`do_down()` (IPython.core.debugger.Pdb method), [231](#)
`do_enable()` (IPython.core.debugger.Pdb method), [231](#)
`do_EOF()` (IPython.core.debugger.Pdb method), [230](#)
`do_exit()` (IPython.core.debugger.Pdb method), [231](#)
`do_h()` (IPython.core.debugger.Pdb method), [231](#)
`do_help()` (IPython.core.debugger.Pdb method), [231](#)
`do_ignore()` (IPython.core.debugger.Pdb method), [231](#)
`do_j()` (IPython.core.debugger.Pdb method), [231](#)
`do_jump()` (IPython.core.debugger.Pdb method), [231](#)
`do_l()` (IPython.core.debugger.Pdb method), [231](#)
`do_list()` (IPython.core.debugger.Pdb method), [231](#)
`do_n()` (IPython.core.debugger.Pdb method), [231](#)
`do_next()` (IPython.core.debugger.Pdb method), [231](#)
`do_p()` (IPython.core.debugger.Pdb method), [231](#)
`do_pdef()` (IPython.core.debugger.Pdb method), [232](#)
`do_pdoc()` (IPython.core.debugger.Pdb method), [232](#)
`do_pinfo()` (IPython.core.debugger.Pdb method), [232](#)
`do_pp()` (IPython.core.debugger.Pdb method), [232](#)
`do_q()` (IPython.core.debugger.Pdb method), [232](#)
`do_quit()` (IPython.core.debugger.Pdb method), [232](#)
`do_r()` (IPython.core.debugger.Pdb method), [232](#)
`do_restart()` (IPython.core.debugger.Pdb method), [232](#)
`do_return()` (IPython.core.debugger.Pdb method), [232](#)
`do_retval()` (IPython.core.debugger.Pdb method), [232](#)
`do_run()` (IPython.core.debugger.Pdb method), [232](#)
`do_rv()` (IPython.core.debugger.Pdb method), [232](#)
`do_s()` (IPython.core.debugger.Pdb method), [232](#)
`do_step()` (IPython.core.debugger.Pdb method), [232](#)
`do_tbreak()` (IPython.core.debugger.Pdb method), [232](#)
`do_u()` (IPython.core.debugger.Pdb method), [232](#)
`do_unalias()` (IPython.core.debugger.Pdb method), [232](#)
`do_unt()` (IPython.core.debugger.Pdb method), [232](#)
`do_until()` (IPython.core.debugger.Pdb method), [232](#)
`do_up()` (IPython.core.debugger.Pdb method), [232](#)
`do_w()` (IPython.core.debugger.Pdb method), [232](#)

`do_what()` (IPython.core.debugger.Pdb method), [232](#)
`do_where()` (IPython.core.debugger.Pdb method), [232](#)
`Doc2UnitTester` (class in IPython.testing.ipunittest), [629](#)
`doctest_ivars()` (in module IPython.testing.plugin.test_refs), [637](#)
`doctest_multiline1()` (in module IPython.testing.plugin.test_ipdoctest), [636](#)
`doctest_multiline2()` (in module IPython.testing.plugin.test_ipdoctest), [636](#)
`doctest_multiline3()` (in module IPython.testing.plugin.test_ipdoctest), [636](#)
`doctest_refs()` (in module IPython.testing.plugin.test_refs), [637](#)
`doctest_reload()` (in module IPython.utils.doctestreload), [650](#)
`doctest_run()` (in module IPython.testing.plugin.test_refs), [637](#)
`doctest_runvars()` (in module IPython.testing.plugin.test_refs), [637](#)
`doctest_simple()` (in module IPython.testing.plugin.test_ipdoctest), [636](#)
`doRemoteCall()` (IPython.kernel.enginefc.FCEngineReferenceFromS method), [462](#)
`doRemoteCall()` (IPython.kernel.enginefc.FCRemoteEngineRefFromS method), [463](#)
`doRemoteCall()` (IPython.kernel.multienginefc.FCSynchronousMulti method), [559](#)
`drop_engine()` (in module IPython.kernel.engineservice), [489](#)

E

`edit()` (IPython.lib.demo.ClearDemo method), [594](#)
`edit()` (IPython.lib.demo.ClearIPDemo method), [596](#)
`edit()` (IPython.lib.demo.Demo method), [598](#)
`edit()` (IPython.lib.demo.IPythonDemo method), [600](#)
`edit()` (IPython.lib.demo.IPythonLineDemo method), [601](#)
`edit()` (IPython.lib.demo.LineDemo method), [603](#)
`EDITOR`, [132](#)
`editor()` (in module IPython.core.hooks), [276](#)
`EmacsChecker` (class in IPython.core.prefilter), [372](#)

- EmacsHandler (class in IPython.core.prefilter), 373
- emptyline() (IPython.core.debugger.Pdb method), 232
- enable_gtk() (IPython.lib.inputhook.InputHookManager method), 607
- enable_gui() (in module IPython.lib.inputhook), 608
- enable_pylab() (IPython.core.interactiveshell.InteractiveShell method), 288
- enable_qt4() (IPython.lib.inputhook.InputHookManager method), 607
- enable_tk() (IPython.lib.inputhook.InputHookManager method), 607
- enable_wx() (IPython.lib.inputhook.InputHookManager method), 608
- enabled (IPython.core.formatters.BaseFormatter attribute), 252
- enabled (IPython.core.formatters.HTMLFormatter attribute), 257
- enabled (IPython.core.formatters.JSONFormatter attribute), 259
- enabled (IPython.core.formatters.LatexFormatter attribute), 261
- enabled (IPython.core.formatters.PlainTextFormatter attribute), 266
- enabled (IPython.core.formatters.PNGFormatter attribute), 263
- enabled (IPython.core.formatters.SVGFormatter attribute), 268
- enabled (IPython.core.prefilter.AliasChecker attribute), 362
- enabled (IPython.core.prefilter.AssignMagicTransformer attribute), 364
- enabled (IPython.core.prefilter.AssignmentChecker attribute), 367
- enabled (IPython.core.prefilter.AssignSystemTransformer attribute), 366
- enabled (IPython.core.prefilter.AutocallChecker attribute), 371
- enabled (IPython.core.prefilter.AutoMagicChecker attribute), 370
- enabled (IPython.core.prefilter.EmacsChecker attribute), 372
- enabled (IPython.core.prefilter.EscCharsChecker attribute), 375
- enabled (IPython.core.prefilter.IPyAutocallChecker attribute), 377
- enabled (IPython.core.prefilter.IPyPromptTransformer attribute), 379
- enabled (IPython.core.prefilter.MultiLineMagicChecker attribute), 382
- enabled (IPython.core.prefilter.PrefilterChecker attribute), 383
- enabled (IPython.core.prefilter.PrefilterTransformer attribute), 389
- enabled (IPython.core.prefilter.PyPromptTransformer attribute), 390
- enabled (IPython.core.prefilter.PythonOpsChecker attribute), 392
- enabled (IPython.core.prefilter.ShellEscapeChecker attribute), 393
- encode (IPython.utils.text.LSString attribute), 675
- endswith (IPython.utils.text.LSString attribute), 675
- EngineAPI (class in IPython.kernel.engineservice), 469
- EngineConnector (class in IPython.kernel.engineconnector), 460
- EngineConnectorError (class in IPython.kernel.engineconnector), 460
- EngineCreationError (class in IPython.kernel.core.error), 435
- EngineError (class in IPython.kernel.core.error), 435
- EngineFromReference (class in IPython.kernel.enginefc), 461
- engineList() (IPython.kernel.multiengine.MultiEngine method), 540
- EngineService (class in IPython.kernel.engineservice), 469
- ensure_fromlist() (in module IPython.lib.deepreload), 590
- environment variable
 - %PATH%, 105
 - EDITOR, 132
 - IPYTHON_DIR, 127
 - PATH, 2
 - PYTHONSTARTUP, 693
- err_text (IPython.kernel.core.output_trap.OutputTrap attribute), 448
- err_text (IPython.kernel.core.redirector_output_trap.RedirectorOutput attribute), 452
- errback() (IPython.kernel.twistedutil.DeferredList method), 582
- error() (in module IPython.utils.warn), 683
- error() (IPython.config.loader.ArgumentParser method), 205
- error() (IPython.core.interactiveshell.SeparateStr method), 322

[error\(\)](#) (IPython.kernel.core.interpreter.Interpreter method), [440](#)
[esc_quotes\(\)](#) (in module IPython.kernel.core.util), [458](#)
[esc_quotes\(\)](#) (in module IPython.utils.text), [681](#)
[esc_strings](#) (IPython.core.prefilter.AliasHandler attribute), [363](#)
[esc_strings](#) (IPython.core.prefilter.AutoHandler attribute), [368](#)
[esc_strings](#) (IPython.core.prefilter.EmacsHandler attribute), [373](#)
[esc_strings](#) (IPython.core.prefilter.HelpHandler attribute), [376](#)
[esc_strings](#) (IPython.core.prefilter.MagicHandler attribute), [381](#)
[esc_strings](#) (IPython.core.prefilter.PrefilterHandler attribute), [385](#)
[esc_strings](#) (IPython.core.prefilter.ShellEscapeHandler attribute), [394](#)
[EscapedTransformer](#) (class in IPython.core.inputsplitters), [278](#)
[EscCharsChecker](#) (class in IPython.core.prefilter), [375](#)
[ev\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [288](#)
[EvalDict](#) (class in IPython.utils.attic), [641](#)
[EventLoopRunner](#) (class in IPython.lib.inputhookwx), [609](#)
[EventLoopTimer](#) (class in IPython.lib.inputhookwx), [609](#)
[EvtHandlerEnabled](#) (IPython.lib.inputhookwx.EventLoopTimer attribute), [610](#)
[ex\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [289](#)
[excepthook\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [289](#)
[exception_colors\(\)](#) (in module IPython.core.excolors), [248](#)
[exclude_aliases\(\)](#) (IPython.core.alias.AliasManager method), [210](#)
[exec_lines\(\)](#) (IPython.kernel.ipengineapp.IEngineApp method), [502](#)
[execRcLines\(\)](#) (IPython.core.debugger.Pdb method), [232](#)
[execute\(\)](#) (IPython.kernel.core.interpreter.Interpreter method), [441](#)
[execute\(\)](#) (IPython.kernel.enginefc.EngineFromReference method), [461](#)
[execute\(\)](#) (IPython.kernel.engineservice.EngineService method), [470](#)
[execute\(\)](#) (IPython.kernel.engineservice.QueuedEngine method), [485](#)
[execute\(\)](#) (IPython.kernel.engineservice.ThreadedEngineService method), [488](#)
[execute\(\)](#) (IPython.kernel.multiengine.MultiEngine method), [540](#)
[execute\(\)](#) (IPython.kernel.multiengine.SynchronousMultiEngine method), [542](#)
[execute\(\)](#) (IPython.kernel.multiengineclient.FullBlockingMultiEngine method), [544](#)
[execute\(\)](#) (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), [557](#)
[execute_block\(\)](#) (IPython.kernel.core.interpreter.Interpreter method), [441](#)
[execute_macro\(\)](#) (IPython.kernel.core.interpreter.Interpreter method), [441](#)
[execute_python\(\)](#) (IPython.kernel.core.interpreter.Interpreter method), [441](#)
[executeAndRaise\(\)](#) (IPython.kernel.engineservice.EngineService method), [470](#)
[executeAndRaise\(\)](#) (IPython.kernel.engineservice.ThreadedEngineService method), [488](#)
[execution_count](#) (IPython.core.interactiveshell.InteractiveShell attribute), [289](#)
[exit\(\)](#) (IPython.config.loader.ArgumentParser method), [205](#)
[exit\(\)](#) (IPython.core.application.Application method), [214](#)
[exit\(\)](#) (IPython.kernel.ipclusterapp.IPClusterApp method), [498](#)
[exit\(\)](#) (IPython.kernel.ipengineapp.IEngineApp method), [502](#)
[exit_now\(\)](#) (IPython.core.interactiveshell.InteractiveShell attribute), [289](#)
[expand_alias\(\)](#) (IPython.core.alias.AliasManager method), [210](#)
[expand_aliases\(\)](#) (IPython.core.alias.AliasManager method), [210](#)
[expand_path\(\)](#) (in module IPython.utils.path), [664](#)
[expand_user\(\)](#) (in module IPython.core.completer), [225](#)
[expandtabs](#) (IPython.utils.text.LSString attribute), [675](#)
[extend](#) (IPython.kernel.core.util.InputList attribute), [457](#)
[extend](#) (IPython.kernel.multiengineclient.QueueStatusList

attribute), 555

extend (IPython.kernel.multiengineclient.ResultList attribute), 556

extend (IPython.utils.text.SList attribute), 679

extends() (IPython.kernel.clientinterfaces.IBlockingClientInterface class method), 421

extends() (IPython.kernel.clientinterfaces.IFCCClientInterface class method), 423

extends() (IPython.kernel.controllerservice.IControllerBase class method), 428

extends() (IPython.kernel.controllerservice.IControllerCore class method), 430

extends() (IPython.kernel.enginefc.IFCCControllerBase class method), 463

extends() (IPython.kernel.enginefc.IFCEngine class method), 466

extends() (IPython.kernel.engineservice.IEngineBase class method), 471

extends() (IPython.kernel.engineservice.IEngineCore class method), 473

extends() (IPython.kernel.engineservice.IEngineProperties class method), 476

extends() (IPython.kernel.engineservice.IEngineQueued class method), 478

extends() (IPython.kernel.engineservice.IEngineSerialized class method), 480

extends() (IPython.kernel.engineservice.IEngineThreaded class method), 483

extends() (IPython.kernel.mapper.IMapper class method), 507

extends() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 509

extends() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

extends() (IPython.kernel.multiengine.IEngineMultiplex class method), 517

extends() (IPython.kernel.multiengine.IFullMultiEngine class method), 519

extends() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 521

extends() (IPython.kernel.multiengine.IMultiEngine class method), 524

extends() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 526

extends() (IPython.kernel.multiengine.IMultiEngineExtras class method), 528

extends() (IPython.kernel.multiengine.ISynchronousEngineMultiplex class method), 531

extends() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 533

extends() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 535

extends() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method), 538

extends() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), 549

extends() (IPython.kernel.multiengineclient.IPendingResult class method), 551

extends() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 560

extends() (IPython.kernel.newserialized.ISerialized class method), 563

extends() (IPython.kernel.newserialized.IUnSerialized class method), 566

extends() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 569

extends() (IPython.kernel.parallelfunction.IParallelFunction class method), 572

extends() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 574

extension_manager (IPython.core.interactiveshell.InteractiveShell attribute), 289

ExtensionManager (class in IPython.core.extensions), 249

extra_args (IPython.core.application.Application attribute), 214

extract_input_slices() (IPython.core.interactiveshell.InteractiveShell method), 289

extract_input_slices() (IPython.core.magic.Magic method), 327

extract_vars() (in module IPython.utils.frame), 651

extract_vars_above() (in module IPython.utils.frame), 651

F

FCEngineReferenceFromService (class in IPython.kernel.enginefc), 462

FCFullSynchronousMultiEngineClient (class in IPython.kernel.multienginefc), 557

FCRemoteEngineRefFromService (class in IPython.kernel.enginefc), 463

FCSynchronousMultiEngineFromMultiEngine (class in IPython.kernel.multienginefc), 559

FDRedirector (class in IPython.lib.pylabtools), 619
 IPython.kernel.core.fd_redirector), 436
 feed_block() (IPython.kernel.core.interpreter.Interpreter method), 441
 fields() (IPython.utils.text.SList method), 679
 figsize() (in module IPython.lib.pylabtools), 619
 figure_to_svg() (in module IPython.lib.pylabtools), 619
 file_config (IPython.core.application.Application attribute), 214
 file_matches() (IPython.core.completer.IPCompleter method), 225
 file_read() (in module IPython.utils.io), 655
 file_readlines() (in module IPython.utils.io), 655
 FileConfigLoader (class in IPython.config.loader), 208
 filefind() (in module IPython.utils.path), 664
 FileLike (class in IPython.kernel.core.file_like), 437
 filename (IPython.core.interactiveshell.InteractiveShell attribute), 289
 FileTimeoutError (class in IPython.kernel.error), 491
 filter() (IPython.utils.wildcard.Namespace method), 685
 find (IPython.utils.text.LSString attribute), 675
 find_cmd() (in module IPython.utils.process), 669
 find_config_file_name() (IPython.core.application.Application method), 214
 find_config_file_name() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
 find_config_file_name() (IPython.kernel.ipengineapp.IPEngineApp method), 502
 find_config_file_paths() (IPython.core.application.Application method), 214
 find_config_file_paths() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
 find_config_file_paths() (IPython.kernel.ipengineapp.IPEngineApp method), 502
 find_cont_furl_file() (IPython.kernel.ipengineapp.IPEngineApp method), 502
 find_gui_and_backend() (in module IPython.lib.pylabtools), 619
 find_handler() (IPython.core.prefilter.PrefilterManager method), 387
 find_head_package() (in module IPython.lib.deepreload), 590
 find_ipython_dir() (IPython.core.application.Application method), 214
 find_ipython_dir() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
 find_ipython_dir() (IPython.kernel.ipengineapp.IPEngineApp method), 502
 find_resources() (IPython.core.application.Application method), 214
 find_resources() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
 find_resources() (IPython.kernel.ipengineapp.IPEngineApp method), 502
 FindCmdError (class in IPython.utils.process), 669
 findsource() (in module IPython.core.ultratb), 410
 finish_cluster_dir() (IPython.kernel.ipclusterapp.IPClusterApp method), 498
 finish_cluster_dir() (IPython.kernel.ipengineapp.IPEngineApp method), 503
 finish_displayhook() (IPython.core.displayhook.DisplayHook method), 241
 finishCommand() (IPython.kernel.engineservice.QueuedEngine method), 485
 fix_error_editor() (in module IPython.core.hooks), 276
 fix_frame_records_filenames() (in module IPython.core.ultratb), 410
 flag_calls() (in module IPython.utils.decorators), 649
 flat_matches() (IPython.utils.strdispatch.StrDispatch method), 670
 flatten() (in module IPython.utils.data), 648
 fload() (IPython.lib.demo.ClearDemo method), 594
 fload() (IPython.lib.demo.ClearIPDemo method), 596
 fload() (IPython.lib.demo.Demo method), 598
 fload() (IPython.lib.demo.IPythonDemo method), 600
 fload() (IPython.lib.demo.IPythonLineDemo method), 601
 fload() (IPython.lib.demo.LineDemo method), 603
 flush() (IPython.core.displayhook.DisplayHook method), 241

flush() (IPython.core.oinspect.myStringIO method), 351

flush() (IPython.kernel.core.fd_redirector.FDRedirector method), 436

flush() (IPython.kernel.core.file_like.FileLike method), 437

flush() (IPython.kernel.core.prompts.CachedOutput method), 449

flush() (IPython.kernel.multiengineclient.FullBlockingMultiEngine method), 545

flush() (IPython.testing.mkdoctests.IndentOutput method), 631

flush() (IPython.utils.io.Tee method), 655

flush_finished() (IPython.lib.backgroundjobs.BackgroundJobManager method), 588

for_type() (IPython.core.formatters.BaseFormatter method), 252

for_type() (IPython.core.formatters.HTMLFormatter method), 257

for_type() (IPython.core.formatters.JSONFormatter method), 259

for_type() (IPython.core.formatters.LatexFormatter method), 261

for_type() (IPython.core.formatters.PlainTextFormatter method), 266

for_type() (IPython.core.formatters.PNGFormatter method), 263

for_type() (IPython.core.formatters.SVGFormatter method), 268

for_type_by_name() (IPython.core.formatters.BaseFormatter method), 252

for_type_by_name() (IPython.core.formatters.HTMLFormatter method), 257

for_type_by_name() (IPython.core.formatters.JSONFormatter method), 259

for_type_by_name() (IPython.core.formatters.LatexFormatter method), 261

for_type_by_name() (IPython.core.formatters.PlainTextFormatter method), 266

for_type_by_name() (IPython.core.formatters.PNGFormatter method), 264

for_type_by_name() (IPython.core.formatters.SVGFormatter method), 269

format() (IPython.core.debugger.Pdb method), 233

format (IPython.utils.text.LSString attribute), 675

format() (IPython.core.formatters.DisplayFormatter method), 254

format() (IPython.utils.PyColorize.Parser method), 640

format() (IPython.core.formatters.SVGFormatter method), 269

format_argspec() (in module IPython.core.oinspect), 352

format_display_data() (in module IPython.core.formatters), 270

format_help() (IPython.config.loader.ArgumentParser method), 205

format_latex() (IPython.core.interactiveshell.InteractiveShell method), 290

format_latex() (IPython.core.magic.Magic method), 327

format_screen() (in module IPython.utils.text), 681

format_stack_entry() (IPython.core.debugger.Pdb method), 233

format_traceback() (IPython.kernel.core.interpreter.Interpreter method), 442

format_type (IPython.core.formatters.BaseFormatter attribute), 253

format_type (IPython.core.formatters.HTMLFormatter attribute), 257

format_type (IPython.core.formatters.JSONFormatter attribute), 259

format_type (IPython.core.formatters.LatexFormatter attribute), 262

format_type (IPython.core.formatters.PlainTextFormatter attribute), 267

format_type (IPython.core.formatters.PNGFormatter attribute), 264

format_type (IPython.core.formatters.SVGFormatter attribute), 269

format_types (IPython.core.formatters.DisplayFormatter attribute), 255

format_usage() (IPython.config.loader.ArgumentParser method), 205

format_version() (IPython.config.loader.ArgumentParser method), 205

FormattedTB (class in IPython.core.ultratb), 404

FormatterABC (class in IPython.core.formatters), 256

formatters (IPython.core.formatters.DisplayFormatter attribute), 255

freeze_term_title() (in module IPython.utils.terminal), 673

fromkeys() (IPython.config.loader.Config static method), 206

fromkeys() (IPython.kernel.core.util.Bunch static method), 456

fromkeys() (IPython.kernel.engineservice.StrictDict static method), 487

fromkeys() (IPython.testing.globalipapp.ipnsdict static method), 625

fromkeys() (IPython.utils.coloransi.ColorSchemeTable static method), 646

fromkeys() (IPython.utils.ipstruct.Struct static method), 658

FrontEndHistory (class in IPython.kernel.core.history), 438

full_path() (in module IPython.testing.tools), 638

FullBlockingMultiEngineClient (class in IPython.kernel.multiengineclient), 543

G

gather() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient static method), 545

gather() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient static method), 557

gatherBoth() (in module IPython.kernel.twistedutil), 583

generate_prompt() (in module IPython.core.hooks), 276

generate_prompt() (IPython.kernel.core.interpreter.Interpreter static method), 442

get (IPython.config.loader.Config attribute), 206

get (IPython.kernel.core.util.Bunch attribute), 456

get (IPython.kernel.engineservice.StrictDict attribute), 487

get (IPython.testing.globalipapp.ipnsdict attribute), 625

get (IPython.utils.coloransi.ColorSchemeTable attribute), 646

get (IPython.utils.ipstruct.Struct attribute), 658

get() (in module IPython.core.ipapi), 324

get() (IPython.core.history.ShadowHist static method), 273

get() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method), 421

get() (IPython.kernel.clientinterfaces.IFCCClientInterfaceProvider class method), 424

get() (IPython.kernel.controllerservice.IControllerBase class method), 428

get() (IPython.kernel.controllerservice.IControllerCore class method), 431

get() (IPython.kernel.enginefc.IFCCControllerBase class method), 464

get() (IPython.kernel.enginefc.IFCEngine class method), 466

get() (IPython.kernel.engineservice.IEngineBase class method), 471

get() (IPython.kernel.engineservice.IEngineCore class method), 474

get() (IPython.kernel.engineservice.IEngineProperties class method), 476

get() (IPython.kernel.engineservice.IEngineQueued class method), 479

get() (IPython.kernel.engineservice.IEngineSerialized class method), 481

get() (IPython.kernel.engineservice.IEngineThreaded class method), 483

get() (IPython.kernel.mapper.IMapper class method), 508

get() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 510

get() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

get() (IPython.kernel.multiengine.IEngineMultiplexer class method), 517

get() (IPython.kernel.multiengine.IFullMultiEngine class method), 520

get() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 522

get() (IPython.kernel.multiengine.IMultiEngine class method), 524

get() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527

get() (IPython.kernel.multiengine.IMultiEngineExtras class method), 529

get() (IPython.kernel.multiengine.ISynchronousEngineMultiplexer class method), 531

get() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 534

get() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 536

get() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method), 539

get() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient (in module class method), 550
 class method), 550
 get() (IPython.kernel.multiengineclient.IPendingResultget_exception_only() (IPython.core.ultratb.AutoFormattedTB class method), 552
 get() (IPython.kernel.multienginefc.IFCSynchronousMultiEnginemethod), 401
 class method), 561
 get_exception_only() (IPython.core.ultratb.ColorTB class method), 403
 get() (IPython.kernel.newserialized.ISerialized class method), 564
 get_exception_only() (IPython.core.ultratb.FormattedTB class method), 566
 get() (IPython.kernel.newserialized.IUnSerialized (IPython.core.ultratb.ListTB class method), 570
 get_exception_only() (IPython.core.ultratb.ParallelDecorator class method), 573
 get_exception_only() (IPython.core.ultratb.ParallelFunction class method), 573
 get() (IPython.kernel.parallelfunction.IParallelFunctionget_exception_only() (IPython.core.ultratb.ParallelFunction class method), 573
 get() (IPython.kernel.parallelfunction.ITaskParallelDecorator 407
 class method), 575
 get_extra_args() (IPython.config.loader.ArgParseConfigLoader class method), 204
 get() (IPython.utils.pickleshare.PickleShareDB method), 667
 get_extra_args() (IPython.core.application.BaseAppConfigLoader class method), 216
 get_all_breaks() (IPython.core.debugger.Pdb method), 233
 get_extra_args() (IPython.kernel.ipclusterapp.IPClusterAppConfigLoader class method), 501
 get_app_qt4() (in module IPython.lib.guisupport), 605
 get_extra_args() (IPython.kernel.ipengineapp.IPEngineAppConfigLoader class method), 505
 get_app_wx() (in module IPython.lib.guisupport), 605
 get_file_breaks() (IPython.core.debugger.Pdb class method), 233
 get_break() (IPython.core.debugger.Pdb method), 233
 get_handler_by_esc() (IPython.core.prefilter.PrefilterManager class method), 387
 get_breaks() (IPython.core.debugger.Pdb method), 233
 get_handler_by_name() (IPython.core.prefilter.PrefilterManager class method), 387
 get_class_members() (in module IPython.utils.dir2), 650
 get_handler_by_name() (IPython.core.prefilter.PrefilterManager class method), 387
 get_client() (IPython.kernel.clientconnector.AsyncClientConnector class method), 411
 get_history() (IPython.core.history.HistoryManager class method), 271
 get_client() (IPython.kernel.clientconnector.ClientConnector class method), 415
 get_history() (IPython.core.interactiveshell.InteractiveShell class method), 290
 get_command() (IPython.kernel.core.interpreter.Interpreter class method), 442
 get_history_item() (IPython.kernel.core.history.FrontEndHistory class method), 438
 get_default() (IPython.config.loader.ArgumentParser class method), 205
 get_history_item() (IPython.kernel.core.history.History class method), 439
 get_default_colors() (in module IPython.core.interactiveshell), 323
 get_history_item() (IPython.kernel.core.history.InterpreterHistory class method), 439
 get_default_value() (IPython.core.interactiveshell.SeparateStr class method), 323
 get_home_dir() (in module IPython.utils.path), 665
 get_deferred_id() (IPython.kernel.multiengine.SynchronousMultiEnginemethod), 542
 get_id() (IPython.kernel.enginefc.EngineFromReference class method), 461
 get_deferred_id() (IPython.kernel.pendingdeferred.PendingDeferred(IPython.kernel.multiengine.MultiEngine class method), 578
 get_id() (IPython.kernel.pendingdeferred.PendingDeferred(IPython.kernel.multiengine.MultiEngine class method), 578
 get_delims() (IPython.core.completer.CompletionSplitter class method), 223
 get_ids() (IPython.kernel.multiengine.SynchronousMultiEngine class method), 542

`get_ids()` (`IPython.kernel.multiengineclient.FullBlockingMultiEngineClient` method), 545
`get_input_after()` (`IPython.kernel.core.history.InterpreterHistory.data()` method), 439
`get_input_cache()` (`IPython.kernel.core.history.InterpreterHistory.engine_client()` method), 439
`get_input_encoding()` (in module `IPython.core.inputsplitter`), 282
`get_ipcluster_logs()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 415
`get_ipcontroller_logs()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 418
`get_ipengine_logs()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 415
`get_ippython()` (in module `IPython.testing.globalipapp`), 626
`get_ipython_dir()` (in module `IPython.utils.path`), 665
`get_ipython_module_path()` (in module `IPython.utils.path`), 665
`get_ipython_package_dir()` (in module `IPython.utils.path`), 665
`get_list()` (`IPython.utils.text.LSString` method), 675
`get_list()` (`IPython.utils.text.SList` method), 680
`get_logs()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 415
`get_logs_by_name()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 415
`get_long_path_name()` (in module `IPython.utils.path`), 665
`get_message_id()` (`IPython.kernel.core.message_cache.IMessageCache` method), 446
`get_messages()` (`IPython.kernel.core.message_cache.SimpleMessageCache` method), 447
`get_history_data()` (`IPython.core.interactiveshell.SeparateStr` method), 323
`get_multiengine_client()` (`IPython.kernel.clientconnector.AsyncClusterConnector.get_multiengine_client()` method), 412
`get_names()` (`IPython.core.debugger.Pdb` method), 233
`get_ns_names()` (`IPython.utils.wildcard.Namespace` method), 685
`get_nlst()` (`IPython.utils.text.SList` method), 680
`get_ns_names()` (`IPython.utils.wildcard.Namespace` method), 685
`get_page_cmd()` (in module `IPython.core.page`), 354
`get_pager_start()` (in module `IPython.core.page`), 354
`get_paths()` (`IPython.utils.text.LSString` method), 675
`get_paths()` (`IPython.utils.text.SList` method), 680
`get_pending_deferred()` (`IPython.kernel.multiengine.SynchronousMultiEngine` method), 542
`get_pending_deferred()` (`IPython.kernel.multiengineclient.FullBlockingMultiEngineClient` method), 545
`get_pending_deferred()` (`IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient` method), 557
`get_pid_from_file()` (`IPython.kernel.ipclusterapp.IPClusterApp` method), 498
`get_pid_from_file()` (`IPython.kernel.ipengineapp.IPEngineApp` method), 498

method), 503

get_plugin() (IPython.core.plugin.PluginManager method), 359

get_properties() (IPython.kernel.enginefc.EngineFromReference method), 461

get_properties() (IPython.kernel.engineservice.EngineService method), 470

get_properties() (IPython.kernel.engineservice.QueuedEngineService method), 485

get_properties() (IPython.kernel.engineservice.ThreadedEngineService method), 488

get_properties() (IPython.kernel.multiengine.MultiEngine method), 540

get_properties() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542

get_properties() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient (in module IPython.core.completerlib), 226

get_properties() (IPython.kernel.multiengineclient.FullSynchronousMultiEngineClient (in module IPython.core.terminal), 673

get_py_filename() (in module IPython.utils.path), 665

get_pyos_inpuhook() (IPython.lib.inpuhook.InputHookManager method), 608

get_pyos_inpuhook_as_func() (IPython.lib.inpuhook.InputHookManager method), 608

get_reference() (IPython.kernel.clientconnector.AsyncClientConnector method), 413

get_result() (IPython.kernel.enginefc.EngineFromReference method), 461

get_result() (IPython.kernel.engineservice.EngineService method), 470

get_result() (IPython.kernel.engineservice.QueuedEngineService method), 485

get_result() (IPython.kernel.engineservice.ThreadedEngineService method), 488

get_result() (IPython.kernel.multiengine.MultiEngine method), 540

get_result() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542

get_result() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 545

get_result() (IPython.kernel.multiengineclient.PendingResult method), 554

get_result() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 557

get_root_modules() (in module IPython.core.terminal), 673

get_slice() (in module IPython.utils.data), 648

get_sptr() (IPython.utils.text.LSString method), 680

get_sptr() (IPython.utils.text.SList method), 680

get_stack() (IPython.core.debugger.Pdb method), 233

get_task_client() (IPython.kernel.clientconnector.AsyncClientConnector method), 413

get_task_client() (IPython.kernel.clientconnector.AsyncCluster method), 415

get_task_client() (IPython.kernel.clientconnector.ClientConnector method), 416

get_task_client() (IPython.kernel.clientconnector.Cluster method), 419

get_task_client() (IPython.kernel.clientconnector.Cluster method), 419

get_bases() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method), 421

get_bases() (IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method), 424

get_bases() (IPython.kernel.controllerservice.IControllerBase class method), 428

get_bases() (IPython.kernel.controllerservice.IControllerCore class method), 431

get_bases() (IPython.kernel.enginefc.IFCControllerBase class method), 464

get_bases() (IPython.kernel.enginefc.IFCEngine class method), 467

get_bases() (IPython.kernel.engineservice.IEngineBase class method), 471

get_bases() (IPython.kernel.engineservice.IEngineCore class method), 474

get_bases() (IPython.kernel.engineservice.IEngineProperties class method), 476

get_bases() (IPython.kernel.engineservice.IEngineQueued class method), 479

get_bases() (IPython.kernel.engineservice.IEngineSerialized class method), 481

get_bases() (IPython.kernel.engineservice.IEngineThreaded class method), 483

get_bases() (IPython.kernel.mapper.IMapper class method), 508

get_bases() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 510

get_bases() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

[getBases\(\) \(IPython.kernel.multiengine.IEngineMultiplexer class method\), 517](#)
[getBases\(\) \(IPython.kernel.multiengine.IFullMultiEngine class method\), 520](#)
[getBases\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine class method\), 522](#)
[getBases\(\) \(IPython.kernel.multiengine.IMultiEngine class method\), 524](#)
[getBases\(\) \(IPython.kernel.multiengine.IMultiEngineCoordinator class method\), 527](#)
[getBases\(\) \(IPython.kernel.multiengine.IMultiEngineExtras class method\), 529](#)
[getBases\(\) \(IPython.kernel.multiengine.ISynchronousEngine class method\), 531](#)
[getBases\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 534](#)
[getBases\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method\), 536](#)
[getBases\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method\), 539](#)
[getBases\(\) \(IPython.kernel.multiengineclient.IFullBlockingClient class method\), 550](#)
[getBases\(\) \(IPython.kernel.multiengineclient.IPendingClient class method\), 552](#)
[getBases\(\) \(IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method\), 561](#)
[getBases\(\) \(IPython.kernel.newserialized.ISerialized class method\), 564](#)
[getBases\(\) \(IPython.kernel.newserialized.IUnSerialized class method\), 566](#)
[getBases\(\) \(IPython.kernel.parallelfunction.IMultiEngineParallelFunction class method\), 570](#)
[getBases\(\) \(IPython.kernel.parallelfunction.IParallelFunction class method\), 573](#)
[getBases\(\) \(IPython.kernel.parallelfunction.ITaskParallelFunction class method\), 575](#)
[GetClassName\(\) \(IPython.lib.inputhookwx.EventLoopFinder class method\), 610](#)
[getData\(\) \(IPython.kernel.newserialized.Serialized class method\), 568](#)
[getData\(\) \(IPython.kernel.newserialized.SerializeIt class method\), 568](#)
[getDataSize\(\) \(IPython.kernel.newserialized.Serialized class method\), 568](#)
[getDataSize\(\) \(IPython.kernel.newserialized.SerializeIt class method\), 568](#)
[getDescriptionFor\(\) \(IPython.kernel.clientinterfaces.IBlockingClient class method\), 421](#)
[getDescriptionFor\(\) \(IPython.kernel.clientinterfaces.IFCCClientInterface class method\), 424](#)
[getDescriptionFor\(\) \(IPython.kernel.controllerservice.IControllerBase class method\), 428](#)
[getDescriptionFor\(\) \(IPython.kernel.controllerservice.IControllerCoordinator class method\), 431](#)
[getDescriptionFor\(\) \(IPython.kernel.enginefc.IFCCControllerBase class method\), 464](#)
[getDescriptionFor\(\) \(IPython.kernel.enginefc.IFCEngine class method\), 467](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineBase class method\), 472](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineCore class method\), 474](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineProperties class method\), 476](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineQueued class method\), 479](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineSerialized class method\), 481](#)
[getDescriptionFor\(\) \(IPython.kernel.engineservice.IEngineThreaded class method\), 483](#)
[getDescriptionFor\(\) \(IPython.kernel.mapper.IMapper class method\), 508](#)
[getDescriptionFor\(\) \(IPython.kernel.mapper.IMultiEngineMapperFactory class method\), 510](#)
[getDescriptionFor\(\) \(IPython.kernel.mapper.ITaskMapperFactory class method\), 512](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IEngineMultiplexer class method\), 517](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IFullMultiEngine class method\), 520](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine class method\), 522](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IMultiEngine class method\), 524](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IMultiEngineCoordinator class method\), 527](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.IMultiEngineExtras class method\), 529](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.ISynchronousEngine class method\), 531](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 534](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method\), 536](#)
[getDescriptionFor\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method\), 539](#)

getDescriptionFor() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient
class method), 550

getDescriptionFor() (IPython.kernel.multiengineclient.IPendingResult class method), 520

getDescriptionFor() (IPython.kernel.multiengineclient.IFullSynchronousMultiEngine
class method), 552

getDescriptionFor() (IPython.kernel.multiengineclient.IFCSynchronousMultiEngine
class method), 561

getDescriptionFor() (IPython.kernel.newserialized.ISerialized class method), 524

getDescriptionFor() (IPython.kernel.newserialized.IUnSerialized class method), 527

getDescriptionFor() (IPython.kernel.newserialized.IUnSerialized class method), 566

getDescriptionFor() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator
class method), 570

getDescriptionFor() (IPython.kernel.parallelfunction.IParallelFunction class method), 531

getDescriptionFor() (IPython.kernel.parallelfunction.IParallelFunction class method), 573

getDescriptionFor() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 534

getDescriptionFor() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575

getdoc() (in module IPython.core.oinspect), 353

getDoc() (IPython.kernel.clientinterfaces.IBlockingClientDecorator class method), 421

getDoc() (IPython.kernel.clientinterfaces.IFCClientInterface class method), 424

getDoc() (IPython.kernel.controllerservice.IControllerBase class method), 428

getDoc() (IPython.kernel.controllerservice.IControllerBase class method), 431

getDoc() (IPython.kernel.engineclient.IFCControllerBase class method), 464

getDoc() (IPython.kernel.engineclient.IFCEngine class method), 467

getDoc() (IPython.kernel.engineservice.IEngineBase class method), 472

getDoc() (IPython.kernel.engineservice.IEngineCore class method), 474

getDoc() (IPython.kernel.engineservice.IEngineProperties class method), 476

getDoc() (IPython.kernel.engineservice.IEngineQueue class method), 479

getDoc() (IPython.kernel.engineservice.IEngineSerialized class method), 481

getDoc() (IPython.kernel.engineservice.IEngineThread class method), 483

getDoc() (IPython.kernel.mapper.IMapper class method), 508

getDoc() (IPython.kernel.mapper.IMultiEngineMapper class method), 510

getDoc() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

getDoc() (IPython.kernel.multiengine.IEngineMultiplex class method), 514

getDoc() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 550

getDoc() (IPython.kernel.multiengineclient.IPendingResult class method), 520

getDoc() (IPython.kernel.multiengineclient.IFullSynchronousMultiEngine class method), 552

getDoc() (IPython.kernel.multiengineclient.IFCSynchronousMultiEngine class method), 561

getDoc() (IPython.kernel.newserialized.ISerialized class method), 524

getDoc() (IPython.kernel.newserialized.IUnSerialized class method), 527

getDoc() (IPython.kernel.newserialized.IUnSerialized class method), 566

getDoc() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 570

getDoc() (IPython.kernel.parallelfunction.IParallelFunction class method), 531

getDoc() (IPython.kernel.parallelfunction.IParallelFunction class method), 573

getDoc() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 534

getDoc() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575

getDoc() (IPython.kernel.controllerservice.IControllerBase class method), 428

getDoc() (IPython.kernel.controllerservice.IControllerBase class method), 431

getDoc() (IPython.kernel.engineclient.IFCControllerBase class method), 464

getDoc() (IPython.kernel.engineclient.IFCEngine class method), 467

getDoc() (IPython.kernel.engineservice.IEngineBase class method), 472

getDoc() (IPython.kernel.engineservice.IEngineCore class method), 474

getDoc() (IPython.kernel.engineservice.IEngineProperties class method), 476

getDoc() (IPython.kernel.engineservice.IEngineQueue class method), 479

getDoc() (IPython.kernel.engineservice.IEngineSerialized class method), 481

getDoc() (IPython.kernel.engineservice.IEngineThread class method), 483

getDoc() (IPython.kernel.mapper.IMapper class method), 508

getDoc() (IPython.kernel.mapper.IMultiEngineMapper class method), 510

getDoc() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

getDoc() (IPython.kernel.multiengine.IEngineMultiplex class method), 514

getDoc() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 550

getDoc() (IPython.kernel.multiengineclient.IPendingResult class method), 520

getDoc() (IPython.kernel.multiengineclient.IFullSynchronousMultiEngine class method), 552

getDoc() (IPython.kernel.multiengineclient.IFCSynchronousMultiEngine class method), 561

getDoc() (IPython.kernel.newserialized.ISerialized class method), 524

getDoc() (IPython.kernel.newserialized.IUnSerialized class method), 527

getDoc() (IPython.kernel.newserialized.IUnSerialized class method), 566

getDoc() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 570

getDoc() (IPython.kernel.parallelfunction.IParallelFunction class method), 531

getDoc() (IPython.kernel.parallelfunction.IParallelFunction class method), 573

getDoc() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 534

getDoc() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575

getEvtHandlerEnabled() (IPython.lib.inputhookwx.EventLoopTimer method), 610

getfigs() (in module IPython.lib.pylabtools), 620

getFunction() (IPython.kernel.pickleutil.CannedFunction method), 579

gethashfile() (in module IPython.utils.pickleshare), 668

getipy (IPython.lib.inputhookwx.EventLoopTimer method), 610

getInterface() (IPython.kernel.engineclient.FCEngineReferenceFromServer method), 462

getInterface() (IPython.kernel.engineclient.FCRemoteEngineRefFromServer method), 462

method), 463

getInterface() (IPython.kernel.multienginefc.FCSynchronousMultiEngine.IEngineMultiplexer
method), 559

getInterfaceName() (IPython.kernel.enginefc.FCEngineRefName() (IPython.service.kernel.multiengine.IFullMultiEngine
method), 462

getInterfaceName() (IPython.kernel.enginefc.FCRemoteEngineRefName() (IPython.service.kernel.multiengine.IFullSynchronousMultiEngine
method), 463

getInterfaceName() (IPython.kernel.multienginefc.FCSynchronousMultiEngineRefName() (IPython.service.kernel.multiengine.IFullSynchronousMultiEngine
method), 559

GetInterval() (IPython.lib.inputhookwx.EventLoopTimer.getName() (IPython.kernel.multiengine.IMultiEngineCoordinator
method), 610

getlink() (IPython.utils.pickleshare.PickleShareDB getName() (IPython.kernel.multiengine.IMultiEngineExtras
method), 667

getMetadata() (IPython.kernel.newserialized.SerializedgetName() (IPython.kernel.multiengine.ISynchronousEngineMultiple
method), 568

getMetadata() (IPython.kernel.newserialized.SerializeIgetName() (IPython.kernel.multiengine.ISynchronousMultiEngine
method), 568

getName() (IPython.core.history.HistorySaveThread getName() (IPython.kernel.multiengine.ISynchronousMultiEngineCo
method), 273

getName() (IPython.kernel.clientinterfaces.IBlockingClientAdapter) (IPython.kernel.multiengine.ISynchronousMultiEngineEx
class method), 421

getName() (IPython.kernel.clientinterfaces.IFCCClientInterfaceName() (IPython.kernel.multiengineclient.IFullBlockingMultiEng
class method), 424

getName() (IPython.kernel.controllerservice.IControllerBase) (IPython.kernel.multiengineclient.IPendingResult
class method), 429

getName() (IPython.kernel.controllerservice.IControllerClient) (IPython.kernel.multienginefc.IFCSynchronousMultiEngi
class method), 431

getName() (IPython.kernel.enginefc.IFCCControllerBase) (IPython.kernel.newserialized.ISerialized
class method), 464

getName() (IPython.kernel.enginefc.IFCEngine getName() (IPython.kernel.newserialized.IUnSerialized
class method), 467

getName() (IPython.kernel.engineservice.IEngineBase) (IPython.kernel.parallelfunction.IMultiEngineParallelDec
class method), 472

getName() (IPython.kernel.engineservice.IEngineCore) (IPython.kernel.parallelfunction.IParallelFunction
class method), 474

getName() (IPython.kernel.engineservice.IEngineProperties) (IPython.kernel.parallelfunction.ITaskParallelDecorator
class method), 476

getName() (IPython.kernel.engineservice.IEngineQueue) (IPython.kernel.twistedutil.ReactorInThread
class method), 479

getName() (IPython.kernel.engineservice.IEngineSerialized) (IPython.lib.backgroundjobs.BackgroundJobBase
class method), 481

getName() (IPython.kernel.engineservice.IEngineThread) (IPython.lib.backgroundjobs.BackgroundJobExpr
class method), 484

getName() (IPython.kernel.mapper.IMapper class) (IPython.lib.backgroundjobs.BackgroundJobFunc
method), 508

getName() (IPython.kernel.mapper.IMultiEngineMapper) (IPython.lib.backgroundjobs.BackgroundJobFunc
class method), 510

getName() (IPython.kernel.mapper.ITaskMapperFactory) (IPython.lib.backgroundjobs.BackgroundJobFunc
method), 510

getNewHandler() (IPython.lib.inputhookwx.EventLoopTimer
method), 610

getObject() (IPython.kernel.newserialized.UnSerialized

method), 568

getObject() (IPython.kernel.newserialized.UnSerializedValue class method), 568

getoutput() (IPython.core.interactiveshell.InteractiveShell class method), 290

getoutputerror() (in module IPython.kernel.core.util), 458

GetOwner() (IPython.lib.inputhookwx.EventLoopTimer class method), 610

getPartition() (IPython.kernel.map.Map class method), 506

getPartition() (IPython.kernel.map.RoundRobinMap class method), 506

GetPreviousHandler() (IPython.lib.inputhookwx.EventLoopTimer class method), 610

getSource() (in module IPython.core.oinspect), 353

getTaggedValue() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method), 421

getTaggedValue() (IPython.kernel.clientinterfaces.IFCClientInterface class method), 424

getTaggedValue() (IPython.kernel.controllerservice.IControllerBase class method), 429

getTaggedValue() (IPython.kernel.controllerservice.IControllerCore class method), 431

getTaggedValue() (IPython.kernel.enginefc.IFCController class method), 464

getTaggedValue() (IPython.kernel.enginefc.IFCEngine class method), 467

getTaggedValue() (IPython.kernel.engineservice.IEngine class method), 472

getTaggedValue() (IPython.kernel.engineservice.IEngine class method), 474

getTaggedValue() (IPython.kernel.engineservice.IEngine class method), 477

getTaggedValue() (IPython.kernel.engineservice.IEngine class method), 479

getTaggedValue() (IPython.kernel.engineservice.IEngineSerialized class method), 481

getTaggedValue() (IPython.kernel.engineservice.IEngineThread class method), 484

getTaggedValue() (IPython.kernel.mapper.IMapper class method), 508

getTaggedValue() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 510

getTaggedValue() (IPython.kernel.mapper.ITaskMapperFactory class method), 512

getTaggedValue() (IPython.kernel.multiengine.IEngine class method), 518

getTaggedValue() (IPython.kernel.multiengine.IFullMultiEngine class method), 520

getTaggedValue() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 522

getTaggedValue() (IPython.kernel.multiengine.IMultiEngine class method), 525

getTaggedValue() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527

getTaggedValue() (IPython.kernel.multiengine.IMultiEngineExtras class method), 529

getTaggedValue() (IPython.kernel.multiengine.ISynchronousEngine class method), 532

getTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 534

getTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 536

getTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 539

getTaggedValue() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), 550

getTaggedValue() (IPython.kernel.multiengineclient.IPendingResult class method), 552

getTaggedValue() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 561

getTaggedValue() (IPython.kernel.newserialized.ISerialized class method), 564

getTaggedValue() (IPython.kernel.newserialized.IUnSerialized class method), 567

getTaggedValue() (IPython.kernel.parallelfunction.IMultiEngineParallelFunction class method), 570

getTaggedValue() (IPython.kernel.parallelfunction.IParallelFunction class method), 573

getTaggedValue() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575

getTaggedValueTags() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method), 422

getTaggedValueTags() (IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method), 424

getTaggedValueTags() (IPython.kernel.controllerservice.IControllerBase class method), 429

getTaggedValueTags() (IPython.kernel.controllerservice.IControllerCore class method), 431

getTaggedValueTags() (IPython.kernel.multiengine.IEngine class method), 518

(IPython.kernel.enginefc.IFCControllerBase class method), 464	(IPython.kernel.multiengine.IMultiEngineExtras class method), 529
getTaggedValueTags() (IPython.kernel.enginefc.IFCEngine class method), 467	getTaggedValueTags() (IPython.kernel.multiengine.ISynchronousEngineMultiplex class method), 532
getTaggedValueTags() (IPython.kernel.engineservice.IEngineBase class method), 472	getTaggedValueTags() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 534
getTaggedValueTags() (IPython.kernel.engineservice.IEngineCore class method), 474	getTaggedValueTags() (IPython.kernel.multiengine.ISynchronousMultiEngineCoord class method), 536
getTaggedValueTags() (IPython.kernel.engineservice.IEngineProperties class method), 477	getTaggedValueTags() (IPython.kernel.multiengine.ISynchronousMultiEngineExtra class method), 539
getTaggedValueTags() (IPython.kernel.engineservice.IEngineQueued class method), 479	getTaggedValueTags() (IPython.kernel.multiengineclient.IFullBlockingMultiEngin class method), 550
getTaggedValueTags() (IPython.kernel.engineservice.IEngineSerialized class method), 481	getTaggedValueTags() (IPython.kernel.multiengineclient.IPendingResult class method), 552
getTaggedValueTags() (IPython.kernel.engineservice.IEngineThreaded class method), 484	getTaggedValueTags() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 561
getTaggedValueTags() (IPython.kernel.mapper.IMapper class method), 508	getTaggedValueTags() (IPython.kernel.newserialized.ISerialized class method), 564
getTaggedValueTags() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 510	getTaggedValueTags() (IPython.kernel.newserialized.IUnSerialized class method), 567
getTaggedValueTags() (IPython.kernel.mapper.ITaskMapperFactory class method), 512	getTaggedValueTags() (IPython.kernel.parallelfunction.IMultiEngineParallelDecor class method), 570
getTaggedValueTags() (IPython.kernel.multiengine.IEngineMultiplexer class method), 518	getTaggedValueTags() (IPython.kernel.parallelfunction.IParallelFunction class method), 573
getTaggedValueTags() (IPython.kernel.multiengine.IFullMultiEngine class method), 520	getTaggedValueTags() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575
getTaggedValueTags() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 522	getTestCaseNames() (in module IPython.testing.nosepatch), 631
getTaggedValueTags() (IPython.kernel.multiengine.IMultiEngine class method), 525	getTypeDescriptor() (IPython.kernel.newserialized.Serialized method), 568
getTaggedValueTags() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527	getTypeDescriptor() (IPython.kernel.newserialized.SerializeIt method), 568
getTaggedValueTags()	getvalue() (IPython.core.oinspect.myStringIO method), 351

[getvalue\(\) \(IPython.kernel.core.fd_redirector.FDRedirector method\), 436](#)
[getvalue\(\) \(IPython.kernel.core.file_like.FileLike method\), 437](#)
[global_matches\(\) \(IPython.core.completer.Completer method\), 222](#)
[global_matches\(\) \(IPython.core.completer.IPCompleter method\), 225](#)
[grep\(\) \(in module IPython.utils.text\), 681](#)
[grep\(\) \(IPython.utils.text.SList method\), 680](#)

H

[handle\(\) \(IPython.core.prefilter.AliasHandler method\), 363](#)
[handle\(\) \(IPython.core.prefilter.AutoHandler method\), 368](#)
[handle\(\) \(IPython.core.prefilter.EmacsHandler method\), 373](#)
[handle\(\) \(IPython.core.prefilter.HelpHandler method\), 376](#)
[handle\(\) \(IPython.core.prefilter.MagicHandler method\), 381](#)
[handle\(\) \(IPython.core.prefilter.PrefilterHandler method\), 385](#)
[handle\(\) \(IPython.core.prefilter.ShellEscapeHandler method\), 394](#)
[handle_command_def\(\) \(IPython.core.debugger.Pdb method\), 233](#)
[handleError\(\) \(IPython.kernel.engineservice.Command method\), 469](#)
[handler\(\) \(IPython.core.ultratb.AutoFormattedTB method\), 401](#)
[handler\(\) \(IPython.core.ultratb.ColorTB method\), 403](#)
[handler\(\) \(IPython.core.ultratb.FormattedTB method\), 404](#)
[handler\(\) \(IPython.core.ultratb.VerboseTB method\), 410](#)
[handler_name \(IPython.core.prefilter.AliasHandler attribute\), 363](#)
[handler_name \(IPython.core.prefilter.AutoHandler attribute\), 368](#)
[handler_name \(IPython.core.prefilter.EmacsHandler attribute\), 373](#)
[handler_name \(IPython.core.prefilter.HelpHandler attribute\), 376](#)
[handler_name \(IPython.core.prefilter.MagicHandler attribute\), 381](#)
[handler_name \(IPython.core.prefilter.PrefilterHandler attribute\), 385](#)
[handler_name \(IPython.core.prefilter.ShellEscapeHandler attribute\), 394](#)
[handleResult\(\) \(IPython.kernel.engineservice.Command method\), 469](#)
[handlers \(IPython.core.prefilter.PrefilterManager attribute\), 387](#)
[has_key \(IPython.kernel.core.util.Bunch attribute\), 456](#)
[has_key \(IPython.kernel.engineservice.StrictDict attribute\), 487](#)
[has_key \(IPython.testing.globalipapp.ipnsdict attribute\), 625](#)
[has_key \(IPython.utils.coloransi.ColorSchemeTable attribute\), 646](#)
[has_key \(IPython.utils.ipstruct.Struct attribute\), 658](#)
[has_key\(\) \(IPython.config.loader.Config method\), 206](#)
[has_key\(\) \(IPython.utils.pickleshare.PickleShareDB method\), 667](#)
[has_magic\(\) \(IPython.kernel.core.magic.Magic method\), 445](#)
[has_open_quotes\(\) \(in module IPython.core.completer\), 226](#)
[has_properties\(\) \(IPython.kernel.enginefc.EngineFromReference method\), 461](#)
[has_properties\(\) \(IPython.kernel.engineservice.EngineService method\), 470](#)
[has_properties\(\) \(IPython.kernel.engineservice.QueuedEngine method\), 485](#)
[has_properties\(\) \(IPython.kernel.engineservice.ThreadedEngineService method\), 488](#)
[has_properties\(\) \(IPython.kernel.multiengine.MultiEngine method\), 541](#)
[has_properties\(\) \(IPython.kernel.multiengine.SynchronousMultiEngine method\), 542](#)
[has_properties\(\) \(IPython.kernel.multiengineclient.FullBlockingMultiEngine method\), 545](#)
[has_properties\(\) \(IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method\), 557](#)
[hasattr\(\) \(IPython.utils.ipstruct.Struct method\), 658](#)
[hcompress\(\) \(IPython.utils.pickleshare.PickleShareDB method\), 667](#)
[hdict\(\) \(IPython.utils.pickleshare.PickleShareDB method\), 667](#)
[help_a\(\) \(IPython.core.debugger.Pdb method\), 233](#)
[help_alias\(\) \(IPython.core.debugger.Pdb method\), 233](#)

- [233](#)
- [help_args\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_b\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_break\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_bt\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_c\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_cl\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_clear\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_commands\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_condition\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_cont\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_continue\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_d\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_debug\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_disable\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_down\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_enable\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_EOF\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_exec\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_exit\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_h\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_help\(\)](#) (IPython.core.debugger.Pdb method), [233](#)
- [help_ignore\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_j\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_jump\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_l\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_list\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_n\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_next\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_p\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_pdb\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_pp\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_q\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_quit\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_r\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_restart\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_return\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_run\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_s\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_step\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_tbreak\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_u\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_unalias\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_unt\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_until\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_up\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_w\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_whatiss\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [help_where\(\)](#) (IPython.core.debugger.Pdb method), [234](#)
- [HelpHandler](#) (class in IPython.core.prefilter), [376](#)
- [hget\(\)](#) (IPython.utils.pickleshare.PickleShareDB method), [667](#)
- [History](#) (class in IPython.kernel.core.history), [438](#)
- [history_length](#) (IPython.core.interactiveshell.InteractiveShell attribute), [290](#)
- [history_manager](#) (IPython.core.interactiveshell.InteractiveShell attribute), [290](#)
- [history_saving_wrapper\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [290](#)
- [HistoryManager](#) (class in IPython.core.history), [271](#)
- [HistorySaveThread](#) (class in IPython.core.history), [273](#)
- [HomeDirError](#) (class in IPython.utils.path), [664](#)
- [hook](#) (IPython.core.display_trap.DisplayTrap attribute), [239](#)

hook() (IPython.kernel.core.display_trap.DisplayTrap method), 434	IEngineProperties (class in IPython.kernel.engineservice), 475	in
hook() (IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method), 454	IFCCClientInterfaceProvider (class in IPython.kernel.clientinterfaces), 423	in
hook() (IPython.kernel.core.traceback_trap.TracebackTrap method), 455	IFCCControllerBase (class in IPython.kernel.enginefc), 463	in
hset() (IPython.utils.pickleshare.PickleShareDB method), 667	IFCEngine (class in IPython.kernel.enginefc), 465	in
HTMLFormatter (class in IPython.core.formatters), 256	IFCSynchronousMultiEngine (class in IPython.kernel.multienginefc), 560	in
I	IFullBlockingMultiEngineClient (class in IPython.kernel.multiengineclient), 549	in
IBlockingClientAdaptor (class in IPython.kernel.clientinterfaces), 420	IFullMultiEngine (class in IPython.kernel.multiengine), 519	in
IControllerBase (class in IPython.kernel.controllerservice), 427	IFullSynchronousMultiEngine (class in IPython.kernel.multiengine), 521	in
IControllerCore (class in IPython.kernel.controllerservice), 430	igrep() (in module IPython.utils.text), 681	
id (IPython.kernel.enginefc.EngineFromReference attribute), 461	IMapper (class in IPython.kernel.mapper), 507	
id (IPython.kernel.engineservice.EngineService attribute), 470	IMessageCache (class in IPython.kernel.core.message_cache), 446	in
id (IPython.kernel.engineservice.ThreadedEngineService attribute), 488	implementedBy() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor static method), 422	
Id (IPython.lib.inputhookwx.EventLoopTimer attribute), 610	implementedBy() (IPython.kernel.clientinterfaces.IFCCClientInterface static method), 424	
ident (IPython.core.history.HistorySaveThread attribute), 273	implementedBy() (IPython.kernel.controllerservice.IControllerBase static method), 429	
ident (IPython.kernel.twistedutil.ReactorInThread attribute), 582	implementedBy() (IPython.kernel.controllerservice.IControllerCore static method), 431	
ident (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 585	implementedBy() (IPython.kernel.enginefc.IFCCControllerBase static method), 464	
ident (IPython.lib.backgroundjobs.BackgroundJobExpire attribute), 586	implementedBy() (IPython.kernel.enginefc.IFCEngine static method), 467	
ident (IPython.lib.backgroundjobs.BackgroundJobFundamental attribute), 587	implementedBy() (IPython.kernel.engineservice.IEngineBase static method), 472	
idgrep() (in module IPython.utils.text), 681	implementedBy() (IPython.kernel.engineservice.IEngineCore static method), 474	
IdInUse (class in IPython.kernel.error), 492	implementedBy() (IPython.kernel.engineservice.IEngineProperties static method), 477	
IDisplayFormatter (class in IPython.kernel.core.display_formatter), 432	implementedBy() (IPython.kernel.engineservice.IEngineQueued static method), 479	
IEngineBase (class in IPython.kernel.engineservice), 470	implementedBy() (IPython.kernel.engineservice.IEngineSerialized static method), 481	
IEngineCore (class in IPython.kernel.engineservice), 473		
IEngineMultiplexer (class in IPython.kernel.multiengine), 516		

[init_display_pub\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_displayhook\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 290
[init_encoding\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 290
[init_environment\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_extension_manager\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_handlers\(\)](#) (IPython.core.prefilter.PrefilterManager method), 387
[init_history\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_hooks\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_inspector\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_instance_attrs\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_io\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_ipython\(\)](#) (in module IPython.core.history), 274
[init_ipython_dir\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_logger\(\)](#) (IPython.core.application.Application method), 214
[init_logger\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_logger\(\)](#) (IPython.kernel.ipclusterapp.IPClusterApp method), 499
[init_logger\(\)](#) (IPython.kernel.ipengineapp.IPEngineApp method), 503
[init_logstart\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_magics\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_payload\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_pdb\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_plugin_manager\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_prefilter\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_prompts\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_pushd_popd_magic\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_sys_modules\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_traceback_handlers\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[init_transformers\(\)](#) (IPython.core.prefilter.PrefilterManager method), 387
[init_validate\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 291
[initialize\(\)](#) (IPython.core.application.Application method), 214
[initialize\(\)](#) (IPython.kernel.ipclusterapp.IPClusterApp method), 503
[initialize\(\)](#) (IPython.kernel.ipengineapp.IPEngineApp method), 503
[input_prefilter\(\)](#) (in module IPython.core.hooks), 276
[input_splitter\(\)](#) (IPython.core.interactiveshell.InteractiveShell attribute), 291
[inputhook_wx1\(\)](#) (in module IPython.lib.inputhookwx), 612
[inputhook_wx2\(\)](#) (in module IPython.lib.inputhookwx), 612
[inputhook_wx3\(\)](#) (in module IPython.lib.inputhookwx), 612
[InputHookManager](#) (class in IPython.lib.inputhook), 606
[InputList](#) (class in IPython.kernel.core.util), 457
[InputSplitter](#) (class in IPython.core.inputsplitters), 280
[InputTermColors](#) (class in IPython.utils.coloransi), 647
[insert](#) (IPython.kernel.core.util.InputList attribute), 458

insert (IPython.kernel.multiengineclient.QueueStatusList attribute), 555

insert (IPython.kernel.multiengineclient.ResultList attribute), 556

insert (IPython.utils.text.SList attribute), 680

inspect_error() (in module IPython.core.ultratb), 410

inspect_object() (in module IPython.utils.generics), 652

Inspector (class in IPython.core.oinspect), 349

install_payload_page() (in module IPython.core.payloadpage), 357

instance() (IPython.core.interactiveshell.InteractiveShell class method), 292

instance_init() (IPython.core.interactiveshell.SeparateStr method), 323

interaction() (IPython.core.debugger.Pdb method), 234

InteractiveMultiEngineClient (class in IPython.kernel.multiengineclient), 553

InteractiveRunner (class in IPython.lib.irunner), 614

InteractiveShell (class in IPython.core.interactiveshell), 285

InteractiveShellABC (class in IPython.core.interactiveshell), 322

interfaces() (IPython.kernel.clientinterfaces.IBlockingClientInterfaces) (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 422

interfaces() (IPython.kernel.clientinterfaces.IFCCClientInterfaces) (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 424

interfaces() (IPython.kernel.controllerservice.IControllerBase) (IPython.kernel.multiengineclient.IPendingResult class method), 429

interfaces() (IPython.kernel.controllerservice.IControllerConf) (IPython.kernel.multiengineclient.IPendingResult class method), 431

interfaces() (IPython.kernel.enginefc.IFCCControllerBase) (IPython.kernel.multiengineclient.IPendingResult class method), 464

interfaces() (IPython.kernel.enginefc.IFCEngine) (IPython.kernel.multiengineclient.IPendingResult class method), 467

interfaces() (IPython.kernel.engineservice.IEngineBase) (IPython.kernel.multiengineclient.IPendingResult class method), 472

interfaces() (IPython.kernel.engineservice.IEngineCore) (IPython.kernel.multiengineclient.IPendingResult class method), 474

interfaces() (IPython.kernel.engineservice.IEngineProperties) (IPython.kernel.multiengineclient.IPendingResult class method), 477

interfaces() (IPython.kernel.engineservice.IEngineQueue) (IPython.kernel.multiengineclient.IPendingResult class method), 479

interfaces() (IPython.kernel.engineservice.IEngineSerialized) (IPython.kernel.multiengineclient.IPendingResult class method), 481

interfaces() (IPython.kernel.engineservice.IEngineThreadLocal) (IPython.kernel.multiengineclient.IPendingResult class method), 484

interfaces() (IPython.kernel.mapper.IMapper class method), 508

interfaces() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 510

interfaces() (IPython.kernel.mapper.ITaskMapperFactory class method), 513

interfaces() (IPython.kernel.multiengine.IEngineMultiplexer class method), 518

interfaces() (IPython.kernel.multiengine.IFullMultiEngine class method), 520

interfaces() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 522

interfaces() (IPython.kernel.multiengine.IMultiEngine class method), 525

interfaces() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527

interfaces() (IPython.kernel.multiengine.IMultiEngineExtras class method), 529

interfaces() (IPython.kernel.multiengine.ISynchronousEngineMultiplexer class method), 532

interfaces() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 534

interfaces() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 537

interfaces() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 539

interfaces() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 550

interfaces() (IPython.kernel.multiengineclient.IPendingResult class method), 552

interfaces() (IPython.kernel.multienginefc.IFCSynchronousMultiEngineClient class method), 561

interfaces() (IPython.kernel.newserialized.ISerialized class method), 564

interfaces() (IPython.kernel.newserialized.IUnSerialized class method), 567

interfaces() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 571

interfaces() (IPython.kernel.parallelfunction.IParallelFunction class method), 573

interfaces() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 575

Interpreter (class in IPython.kernel.core.interpreter), 440

InterpreterHistory (class in IPython.kernel.core.history), 439

Interval (IPython.lib.inputhookwx.EventLoopTimer attribute), 610

- InvalidAliasError (class in IPython.core.alias), 212
- InvalidClientID (class in IPython.kernel.error), 492
- InvalidDeferredID (class in IPython.kernel.error), 492
- InvalidEngineID (class in IPython.kernel.error), 492
- InvalidProperty (class in IPython.kernel.error), 493
- IOStream (class in IPython.utils.io), 654
- IOTerm (class in IPython.utils.io), 654
- IParallelFunction (class in IPython.kernel.parallelfunction), 572
- IPClusterApp (class in IPython.kernel.ipclusterapp), 498
- IPClusterAppConfigLoader (class in IPython.kernel.ipclusterapp), 500
- IPCompleter (class in IPython.core.completer), 223
- ipdocstring() (in module IPython.testing.ipunittest), 629
- IPendingResult (class in IPython.kernel.multiengineclient), 551
- IPEngineApp (class in IPython.kernel.ipengineapp), 502
- IPEngineAppConfigLoader (class in IPython.kernel.ipengineapp), 504
- ipexec() (in module IPython.testing.tools), 639
- ipexec_validate() (in module IPython.testing.tools), 639
- ipfunc() (in module IPython.testing.plugin.dtexample), 632
- ipmagic() (IPython.kernel.core.interpreter.Interpreter method), 442
- ipnsdict (class in IPython.testing.globalipapp), 625
- iprand() (in module IPython.testing.plugin.dtexample), 633
- iprand_all() (in module IPython.testing.plugin.dtexample), 633
- ipsystem() (IPython.kernel.core.interpreter.Interpreter method), 443
- IPTester (class in IPython.testing.ipctest), 627
- IPyAutocall (class in IPython.core.autocall), 217
- IPyAutocallChecker (class in IPython.core.prefilter), 377
- ipyfunc2() (in module IPython.testing.plugin.simple), 635
- IPyPromptTransformer (class in IPython.core.prefilter), 378
- IPython.config.loader (module), 203
- IPython.core.alias (module), 209
- IPython.core.application (module), 212
- IPython.core.autocall (module), 217
- IPython.core.builtin_trap (module), 218
- IPython.core.compilerop (module), 220
- IPython.core.completer (module), 221
- IPython.core.completerlib (module), 226
- IPython.core.crashhandler (module), 227
- IPython.core.debugger (module), 229
- IPython.core.display (module), 237
- IPython.core.display_trap (module), 239
- IPython.core.displayhook (module), 240
- IPython.core.displaypub (module), 243
- IPython.core.error (module), 246
- IPython.core.excolors (module), 248
- IPython.core.extensions (module), 248
- IPython.core.formatters (module), 251
- IPython.core.history (module), 271
- IPython.core.hooks (module), 275
- IPython.core.inputsplitter (module), 277
- IPython.core.interactiveshell (module), 284
- IPython.core.ipapi (module), 324
- IPython.core.logger (module), 324
- IPython.core.macro (module), 325
- IPython.core.magic (module), 326
- IPython.core.oinspect (module), 348
- IPython.core.page (module), 353
- IPython.core.payload (module), 355
- IPython.core.payloadpage (module), 356
- IPython.core.plugin (module), 357
- IPython.core.prefilter (module), 361
- IPython.core.prompts (module), 396
- IPython.core.splitinput (module), 399
- IPython.core.ultratb (module), 399
- IPython.kernel.clientconnector (module), 411
- IPython.kernel.clientinterfaces (module), 420
- IPython.kernel.codeutil (module), 425
- IPython.kernel.controllerservice (module), 426
- IPython.kernel.core.display_formatter (module), 432
- IPython.kernel.core.display_trap (module), 433
- IPython.kernel.core.error (module), 434
- IPython.kernel.core.fd_redirector (module), 436
- IPython.kernel.core.file_like (module), 437
- IPython.kernel.core.history (module), 438
- IPython.kernel.core.interpreter (module), 440
- IPython.kernel.core.macro (module), 444
- IPython.kernel.core.magic (module), 445
- IPython.kernel.core.message_cache (module), 446
- IPython.kernel.core.output_trap (module), 447

- IPython.kernel.core.prompts (module), 448
- IPython.kernel.core.redirector_output_trap (module), 452
- IPython.kernel.core.sync_traceback_trap (module), 453
- IPython.kernel.core.traceback_formatter (module), 454
- IPython.kernel.core.traceback_trap (module), 455
- IPython.kernel.core.util (module), 456
- IPython.kernel.engineconnector (module), 459
- IPython.kernel.enginefc (module), 460
- IPython.kernel.engineservice (module), 468
- IPython.kernel.error (module), 490
- IPython.kernel.ipclusterapp (module), 498
- IPython.kernel.ipengineapp (module), 501
- IPython.kernel.map (module), 505
- IPython.kernel.mapper (module), 506
- IPython.kernel.multiengine (module), 515
- IPython.kernel.multiengineclient (module), 543
- IPython.kernel.multienginefc (module), 556
- IPython.kernel.newserialized (module), 563
- IPython.kernel.parallelfunction (module), 569
- IPython.kernel.pbutil (module), 577
- IPython.kernel.pendingdeferred (module), 578
- IPython.kernel.pickleutil (module), 579
- IPython.kernel.twistedutil (module), 580
- IPython.kernel.util (module), 584
- IPython.lib.backgroundjobs (module), 585
- IPython.lib.clipboard (module), 589
- IPython.lib.deeppreload (module), 590
- IPython.lib.demo (module), 591
- IPython.lib.guisupport (module), 604
- IPython.lib.inputhook (module), 606
- IPython.lib.inputhookwx (module), 609
- IPython.lib.irunner (module), 613
- IPython.lib.latextools (module), 618
- IPython.lib.pylabtools (module), 619
- IPython.testing (module), 621
- IPython.testing.decorators (module), 621
- IPython.testing.decorators_trial (module), 623
- IPython.testing.globalipapp (module), 624
- IPython.testing.iptest (module), 626
- IPython.testing.ipunittest (module), 628
- IPython.testing.mkdoctests (module), 630
- IPython.testing.nosepatch (module), 631
- IPython.testing.parametric (module), 632
- IPython.testing.plugin.dtexample (module), 632
- IPython.testing.plugin.show_refs (module), 634
- IPython.testing.plugin.simple (module), 635
- IPython.testing.plugin.test_ipdoctest (module), 635
- IPython.testing.plugin.test_refs (module), 636
- IPython.testing.tools (module), 637
- IPython.utils.attic (module), 641
- IPython.utils.autoattr (module), 643
- IPython.utils.coloransi (module), 645
- IPython.utils.data (module), 648
- IPython.utils.decorators (module), 649
- IPython.utils.dir2 (module), 649
- IPython.utils.doctestreload (module), 650
- IPython.utils.frame (module), 651
- IPython.utils.generics (module), 651
- IPython.utils.growl (module), 652
- IPython.utils.importstring (module), 653
- IPython.utils.io (module), 654
- IPython.utils.ipstruct (module), 656
- IPython.utils.jsonutil (module), 660
- IPython.utils.notification (module), 661
- IPython.utils.path (module), 663
- IPython.utils.pickleshare (module), 666
- IPython.utils.process (module), 668
- IPython.utils.PyColorize (module), 640
- IPython.utils.strdispatch (module), 670
- IPython.utils.sysinfo (module), 671
- IPython.utils.syspathcontext (module), 672
- IPython.utils.terminal (module), 673
- IPython.utils.text (module), 674
- IPython.utils.timing (module), 682
- IPython.utils.upgradedir (module), 683
- IPython.utils.warn (module), 683
- IPython.utils.wildcard (module), 684
- IPython2PythonConverter (class in IPython.testing.ipunittest), 629
- IPYTHON_DIR, 127
- ipython_dir (IPython.core.application.Application attribute), 215
- ipython_dir (IPython.core.interactiveshell.InteractiveShell attribute), 292
- ipython_extension_dir (IPython.core.extensions.ExtensionManager attribute), 249
- IPythonCoreError (class in IPython.core.error), 247
- IPythonDemo (class in IPython.lib.demo), 599
- IPythonError (class in IPython.kernel.core.error), 435
- IPythonGrowlError (class in IPython.utils.growl), 652

[IPythonInputSplitter](#) (class in [IPython.core.inputsplitters](#)), [278](#)
[IPythonLineDemo](#) (class in [IPython.lib.demo](#)), [601](#)
[IPythonRunner](#) (class in [IPython.lib.irunner](#)), [613](#)
[is_alive\(\)](#) ([IPython.core.history.HistorySaveThread](#) class method), [273](#)
[is_alive\(\)](#) ([IPython.kernel.twistedutil.ReactorInThread](#) class method), [583](#)
[is_alive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobBase](#) class method), [585](#)
[is_alive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobExpr](#) class method), [586](#)
[is_alive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobFunc](#) class method), [587](#)
[is_event_loop_running_qt4\(\)](#) (in module [IPython.lib.guisupport](#)), [605](#)
[is_event_loop_running_wx\(\)](#) (in module [IPython.lib.guisupport](#)), [605](#)
[is_importable\(\)](#) (in module [IPython.core.completerlib](#)), [226](#)
[is_shadowed\(\)](#) (in module [IPython.core.prefilter](#)), [395](#)
[is_type\(\)](#) (in module [IPython.utils.wildcard](#)), [685](#)
[isAlive\(\)](#) ([IPython.core.history.HistorySaveThread](#) class method), [273](#)
[isAlive\(\)](#) ([IPython.kernel.twistedutil.ReactorInThread](#) class method), [582](#)
[isAlive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobBase](#) class method), [585](#)
[isAlive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobExpr](#) class method), [586](#)
[isAlive\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobFunc](#) class method), [587](#)
[isalnum](#) ([IPython.utils.text.LSString](#) attribute), [675](#)
[isalpha](#) ([IPython.utils.text.LSString](#) attribute), [676](#)
[isatty\(\)](#) ([IPython.core.oinspect.myStringIO](#) method), [351](#)
[isatty\(\)](#) ([IPython.kernel.core.file_like.FileLike](#) class method), [437](#)
[isDaemon\(\)](#) ([IPython.core.history.HistorySaveThread](#) class method), [273](#)
[isDaemon\(\)](#) ([IPython.kernel.twistedutil.ReactorInThread](#) class method), [583](#)
[isDaemon\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobBase](#) class method), [585](#)
[isDaemon\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobExpr](#) class method), [586](#)
[isDaemon\(\)](#) ([IPython.lib.backgroundjobs.BackgroundJobFunc](#) class method), [587](#)
[isdigit](#) ([IPython.utils.text.LSString](#) attribute), [676](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.clientinterfaces.IBlockingClientAdaptor](#) class method), [422](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.clientinterfaces.IFCCClientInterfaceProvider](#) class method), [424](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.controllerservice.IControllerBase](#) class method), [429](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.controllerservice.IControllerCore](#) class method), [431](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.enginefc.IFCCControllerBase](#) class method), [465](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.enginefc.IFCEngine](#) class method), [467](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineBase](#) class method), [472](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineCore](#) class method), [475](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineProperties](#) class method), [477](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineQueued](#) class method), [479](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineSerialized](#) class method), [482](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.engineservice.IEngineThreaded](#) class method), [484](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.mapper.IMapper](#) class method), [508](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.mapper.IMultiEngineMapperFactory](#) class method), [511](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.mapper.ITaskMapperFactory](#) class method), [513](#)
[isEqualOrExtendedBy\(\)](#) ([IPython.kernel.mapper.ITaskMapperFactory](#) class method), [513](#)

(IPython.kernel.multiengine.IEngineMultiplexer class method), 518	(IPython.kernel.parallelfunction.IParallelFunction class method), 573
isEqualOrExtendedBy() (IPython.kernel.multiengine.IFullMultiEngine class method), 520	isEqualOrExtendedBy() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 576
isEqualOrExtendedBy() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 523	ISerialized (class in IPython.kernel.newserialized), 566
isEqualOrExtendedBy() (IPython.kernel.multiengine.IMultiEngine class method), 525	islower (IPython.utils.text.LSString attribute), 676
isEqualOrExtendedBy() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527	IsOneShot() (IPython.lib.inputhookwx.EventLoopTimer method), 611
isEqualOrExtendedBy() (IPython.kernel.multiengine.IMultiEngineExtras class method), 530	isOrExtends() (IPython.kernel.clientinterfaces.IBlockingClientAdapter static method), 422
isEqualOrExtendedBy() (IPython.kernel.multiengine.ISynchronousEngine class method), 532	isOrExtends() (IPython.kernel.clientinterfaces.IFCCClientInterfaceProxy static method), 424
isEqualOrExtendedBy() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 534	isOrExtends() (IPython.kernel.controllerservice.IControllerBase static method), 429
isEqualOrExtendedBy() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 537	isOrExtends() (IPython.kernel.controllerservice.IControllerCore static method), 431
isEqualOrExtendedBy() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method), 539	isOrExtends() (IPython.kernel.enginefc.IFCCControllerBase static method), 465
isEqualOrExtendedBy() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine class method), 550	isOrExtends() (IPython.kernel.enginefc.IFCEngine static method), 467
isEqualOrExtendedBy() (IPython.kernel.multiengineclient.IPendingResult class method), 553	isOrExtends() (IPython.kernel.engineservice.IEngineBase static method), 472
isEqualOrExtendedBy() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 562	isOrExtends() (IPython.kernel.engineservice.IEngineCore static method), 475
isEqualOrExtendedBy() (IPython.kernel.newserialized.ISerialized class method), 565	isOrExtends() (IPython.kernel.engineservice.IEngineProperties static method), 477
isEqualOrExtendedBy() (IPython.kernel.newserialized.IUnSerialized class method), 567	isOrExtends() (IPython.kernel.engineservice.IEngineQueued static method), 479
isEqualOrExtendedBy() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 571	isOrExtends() (IPython.kernel.engineservice.IEngineSerialized static method), 482
	isOrExtends() (IPython.kernel.engineservice.IEngineThreaded static method), 484
	isOrExtends() (IPython.kernel.mapper.IMapper static method), 508
	isOrExtends() (IPython.kernel.mapper.IMultiEngineMapperFactory static method), 511
	isOrExtends() (IPython.kernel.mapper.ITaskMapperFactory static method), 513
	isOrExtends() (IPython.kernel.multiengine.IEngineMultiplexer static method), 518
	isOrExtends() (IPython.kernel.multiengine.IFullMultiEngine static method), 520
	isOrExtends() (IPython.kernel.multiengine.IFullSynchronousMultiEngine static method), 523
	isOrExtends() (IPython.kernel.multiengine.IMultiEngine static method), 525

isOrExtends() (IPython.kernel.multiengine.IMultiEngineCoordinator method), 527

isOrExtends() (IPython.kernel.multiengine.IMultiEngineExtras static method), 530

isOrExtends() (IPython.kernel.multiengine.ISynchronousEngineMultiplexer static method), 532

isOrExtends() (IPython.kernel.multiengine.ISynchronousMultiEngine static method), 534

isOrExtends() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator static method), 537

isOrExtends() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras static method), 539

isOrExtends() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient static method), 550

isOrExtends() (IPython.kernel.multiengineclient.IPendingResult static method), 553

isOrExtends() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine static method), 562

isOrExtends() (IPython.kernel.newserialized.ISerializedIteritems static method), 565

isOrExtends() (IPython.kernel.newserialized.IUnSerializedIteritems static method), 567

isOrExtends() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator static method), 571

isOrExtends() (IPython.kernel.parallelfunction.IParallelFunction static method), 573

isOrExtends() (IPython.kernel.parallelfunction.ITaskParallelDecorator static method), 576

IsRunning() (IPython.lib.inputhookwx.EventLoopTimer method), 611

IsSameAs() (IPython.lib.inputhookwx.EventLoopTimer method), 611

isspace (IPython.utils.text.LSString attribute), 676

istitle (IPython.utils.text.LSString attribute), 676

IsUnlinked() (IPython.lib.inputhookwx.EventLoopTimer method), 611

isupper (IPython.utils.text.LSString attribute), 676

ISynchronousEngineMultiplexer (class in IPython.kernel.multiengine), 530

ISynchronousMultiEngine (class in IPython.kernel.multiengine), 533

ISynchronousMultiEngineCoordinator (class in IPython.kernel.multiengine), 535

ISynchronousMultiEngineExtras (class in IPython.kernel.multiengine), 538

ITaskMapperFactory (class in IPython.kernel.mapper), 511

ITaskParallelDecorator (class in IPython.kernel.parallelfunction), 574

items (IPython.config.loader.Config attribute), 206

items (IPython.kernel.core.util.Bunch attribute), 456

items (IPython.kernel.engineservice.StrictDict attribute), 487

items (IPython.testing.globalipapp.ipnsdict attribute), 625

items (IPython.utils.coloransi.ColorSchemeTable attribute), 646

items (IPython.utils.ipstruct.Struct attribute), 658

items (IPython.utils.pickleshare.PickleShareDB attribute), 667

items (IPython.config.loader.Config attribute), 206

items (IPython.kernel.core.util.Bunch attribute), 456

items (IPython.kernel.engineservice.StrictDict attribute), 487

items (IPython.testing.globalipapp.ipnsdict attribute), 625

items (IPython.utils.coloransi.ColorSchemeTable attribute), 646

items (IPython.utils.ipstruct.Struct attribute), 658

items (IPython.utils.pickleshare.PickleShareDB attribute), 667

items (IPython.config.loader.Config attribute), 206

items (IPython.kernel.core.util.Bunch attribute), 456

items (IPython.kernel.engineservice.StrictDict attribute), 487

items (IPython.testing.globalipapp.ipnsdict attribute), 625

items (IPython.utils.coloransi.ColorSchemeTable attribute), 647

- ul style="list-style-type: none; padding-left: 0;">
- itervalues (IPython.utils.ipstruct.Struct attribute), 658
- itervalues() (IPython.utils.pickleshare.PickleShareDB method), 667
- ITracebackFormatter (class in IPython.kernel.core.traceback_formatter), 454
- IUnSerialized (class in IPython.kernel.newserialized), 565
- J**
- join (IPython.utils.text.LSString attribute), 676
- join() (IPython.core.history.HistorySaveThread method), 273
- join() (IPython.kernel.twistedutil.ReactorInThread method), 583
- join() (IPython.lib.backgroundjobs.BackgroundJobBase method), 586
- join() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586
- join() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587
- joinPartitions() (IPython.kernel.map.Map method), 506
- joinPartitions() (IPython.kernel.map.RoundRobinMap method), 506
- json_clean() (in module IPython.utils.jsonutil), 660
- JSONFormatter (class in IPython.core.formatters), 258
- jump() (IPython.lib.demo.ClearDemo method), 594
- jump() (IPython.lib.demo.ClearIPDemo method), 596
- jump() (IPython.lib.demo.Demo method), 598
- jump() (IPython.lib.demo.IPythonDemo method), 600
- jump() (IPython.lib.demo.IPythonLineDemo method), 601
- jump() (IPython.lib.demo.LineDemo method), 603
- K**
- KernelError (class in IPython.kernel.error), 493
- keys (IPython.config.loader.Config attribute), 206
- keys (IPython.kernel.core.util.Bunch attribute), 457
- keys (IPython.kernel.engineservice.StrictDict attribute), 487
- keys (IPython.testing.globalipapp.ipnsdict attribute), 625
- keys (IPython.utils.coloransi.ColorSchemeTable attribute), 647
- keys (IPython.utils.ipstruct.Struct attribute), 658
- keys() (IPython.kernel.enginefc.EngineFromReference method), 461
- keys() (IPython.kernel.engineservice.EngineService method), 470
- keys() (IPython.kernel.engineservice.QueuedEngine method), 485
- keys() (IPython.kernel.engineservice.ThreadedEngineService method), 488
- keys() (IPython.kernel.multiengine.MultiEngine method), 541
- keys() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542
- keys() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 545
- keys() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), 557
- keys() (IPython.utils.pickleshare.PickleShareDB method), 667
- kill() (IPython.kernel.enginefc.EngineFromReference method), 461
- kill() (IPython.kernel.engineservice.EngineService method), 470
- kill() (IPython.kernel.engineservice.QueuedEngine method), 485
- kill() (IPython.kernel.engineservice.ThreadedEngineService method), 488
- kill() (IPython.kernel.multiengine.MultiEngine method), 541
- kill() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542
- kill() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 545
- kill() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), 557
- killBack() (IPython.kernel.enginefc.EngineFromReference method), 461
- L**
- l (IPython.utils.text.LSString attribute), 676
- l (IPython.utils.text.SList attribute), 680
- late_startup_hook() (in module IPython.core.hooks), 276
- latex_to_html() (in module IPython.lib.latextools), 619

[latex_to_png\(\)](#) (in module `IPython.lib.latextools`), [619](#)
[LatexFormatter](#) (class in `IPython.core.formatters`), [260](#)
[launch_new_instance\(\)](#) (in module `IPython.kernel.ipclusterapp`), [501](#)
[launch_new_instance\(\)](#) (in module `IPython.kernel.ipengineapp`), [505](#)
[LineDemo](#) (class in `IPython.lib.demo`), [602](#)
[LineInfo](#) (class in `IPython.core.inputsplitter`), [282](#)
[LineInfo](#) (class in `IPython.core.prefilter`), [380](#)
[lineinfo\(\)](#) (`IPython.core.debugger.Pdb` method), [234](#)
[list](#) (`IPython.utils.text.LSString` attribute), [676](#)
[list](#) (`IPython.utils.text.SList` attribute), [680](#)
[list2dict\(\)](#) (in module `IPython.utils.data`), [648](#)
[list2dict2\(\)](#) (in module `IPython.utils.data`), [648](#)
[list_cluster_dirs\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[list_command_pydb\(\)](#) (`IPython.core.debugger.Pdb` method), [234](#)
[list_namespace\(\)](#) (in module `IPython.utils.wildcard`), [685](#)
[list_strings\(\)](#) (in module `IPython.utils.text`), [681](#)
[ListTB](#) (class in `IPython.core.ultratb`), [405](#)
[ljust](#) (`IPython.utils.text.LSString` attribute), [676](#)
[load_command_line_config\(\)](#) (`IPython.core.application.Application` method), [215](#)
[load_command_line_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[load_command_line_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [503](#)
[load_config\(\)](#) (`IPython.config.loader.ArgParseConfigLoader` method), [204](#)
[load_config\(\)](#) (`IPython.config.loader.CommandLineConfigLoader` method), [205](#)
[load_config\(\)](#) (`IPython.config.loader.ConfigLoader` method), [207](#)
[load_config\(\)](#) (`IPython.config.loader.FileConfigLoader` method), [208](#)
[load_config\(\)](#) (`IPython.config.loader.PyFileConfigLoader` method), [209](#)
[load_config\(\)](#) (`IPython.core.application.BaseAppConfigLoader` method), [216](#)
[load_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [501](#)
[load_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [505](#)
[load_extension\(\)](#) (`IPython.core.extensions.ExtensionManager` method), [249](#)
[load_file_config\(\)](#) (`IPython.core.application.Application` method), [215](#)
[load_file_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[load_file_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [503](#)
[load_tail\(\)](#) (in module `IPython.lib.deepreload`), [590](#)
[location](#) (`IPython.kernel.clientconnector.AsyncCluster` attribute), [415](#)
[location](#) (`IPython.kernel.clientconnector.Cluster` attribute), [419](#)
[log\(\)](#) (`IPython.core.logger.Logger` method), [324](#)
[log_command_line_config\(\)](#) (`IPython.core.application.Application` method), [215](#)
[log_command_line_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[log_command_line_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [503](#)
[log_default_config\(\)](#) (`IPython.core.application.Application` method), [215](#)
[log_default_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[log_default_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [503](#)
[log_err\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[log_file_config\(\)](#) (`IPython.core.application.Application` method), [215](#)
[log_file_config\(\)](#) (`IPython.kernel.ipclusterapp.IPClusterApp` method), [499](#)
[log_file_config\(\)](#) (`IPython.kernel.ipengineapp.IPEngineApp` method), [503](#)
[log_level](#) (`IPython.core.application.Application` attribute), [215](#)
[log_level](#) (`IPython.kernel.ipclusterapp.IPClusterApp` attribute), [499](#)
[log_level](#) (`IPython.kernel.ipengineapp.IPEngineApp` attribute), [503](#)

[log_master_config\(\)](#) (IPython.core.application.Application method), 215
[log_master_config\(\)](#) (IPython.kernel.ipclusterapp.IPClusterApp method), 499
[log_master_config\(\)](#) (IPython.kernel.ipengineapp.IPEngineApp method), 503
[log_output\(\)](#) (IPython.core.displayhook.DisplayHook method), 241
[log_write\(\)](#) (IPython.core.logger.Logger method), 325
[logappend](#) (IPython.core.interactiveshell.InteractiveShell attribute), 292
[logfile](#) (IPython.core.interactiveshell.InteractiveShell attribute), 292
[Logger](#) (class in IPython.core.logger), 324
[logmode](#) (IPython.core.logger.Logger attribute), 325
[logstart](#) (IPython.core.interactiveshell.InteractiveShell attribute), 292
[logstart\(\)](#) (IPython.core.logger.Logger method), 325
[logstate\(\)](#) (IPython.core.logger.Logger method), 325
[logstop\(\)](#) (IPython.core.logger.Logger method), 325
[lookupmodule\(\)](#) (IPython.core.debugger.Pdb method), 234
[lower](#) (IPython.utils.text.LSString attribute), 676
[lsmagic\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 292
[lsmagic\(\)](#) (IPython.core.magic.Magic method), 327
[LSString](#) (class in IPython.utils.text), 674
[lstrip](#) (IPython.utils.text.LSString attribute), 677

M

[Macro](#) (class in IPython.core.macro), 326
[Macro](#) (class in IPython.kernel.core.macro), 445
[Magic](#) (class in IPython.core.magic), 326
[Magic](#) (class in IPython.kernel.core.magic), 445
[magic\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 292
[magic_alias\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 292
[magic_alias\(\)](#) (IPython.core.magic.Magic method), 327
[magic_autocall\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 293
[magic_autocall\(\)](#) (IPython.core.magic.Magic method), 328
[magic_automagic\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 294
[magic_automagic\(\)](#) (IPython.core.magic.Magic method), 329
[magic_bookmark\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 294
[magic_bookmark\(\)](#) (IPython.core.magic.Magic method), 329
[magic_cd\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 294
[magic_cd\(\)](#) (IPython.core.magic.Magic method), 329
[magic_colors\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 295
[magic_colors\(\)](#) (IPython.core.magic.Magic method), 330
[magic_debug\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 295
[magic_debug\(\)](#) (IPython.core.magic.Magic method), 330
[magic_dhist\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 295
[magic_dhist\(\)](#) (IPython.core.magic.Magic method), 330
[magic_dirs\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 295
[magic_dirs\(\)](#) (IPython.core.magic.Magic method), 330
[magic_doctest_mode\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 295
[magic_doctest_mode\(\)](#) (IPython.core.magic.Magic method), 330
[magic_ed\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 296
[magic_ed\(\)](#) (IPython.core.magic.Magic method), 331
[magic_edit\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 296
[magic_edit\(\)](#) (IPython.core.magic.Magic method), 331
[magic_env\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 298
[magic_env\(\)](#) (IPython.core.magic.Magic method), 333
[magic_env\(\)](#) (IPython.kernel.core.magic.Magic method), 445
[magic_Exit\(\)](#) (IPython.core.interactiveshell.InteractiveShell

method), 292

magic_exit() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_Exit() (IPython.core.magic.Magic method), 327

magic_exit() (IPython.core.magic.Magic method), 333

magic_gui() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_gui() (IPython.core.magic.Magic method), 333

magic_hist() (in module IPython.core.history), 274

magic_history() (in module IPython.core.history), 274

magic_install_default_config() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_install_default_config() (IPython.core.magic.Magic method), 333

magic_install_profiles() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_install_profiles() (IPython.core.magic.Magic method), 333

magic_load_ext() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_load_ext() (IPython.core.magic.Magic method), 333

magic_logoff() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_logoff() (IPython.core.magic.Magic method), 333

magic_logon() (IPython.core.interactiveshell.InteractiveShell method), 298

magic_logon() (IPython.core.magic.Magic method), 333

magic_logstart() (IPython.core.interactiveshell.InteractiveShell method), 299

magic_logstart() (IPython.core.magic.Magic method), 334

magic_logstate() (IPython.core.interactiveshell.InteractiveShell method), 299

magic_logstate() (IPython.core.magic.Magic method), 334

magic_logstop() (IPython.core.interactiveshell.InteractiveShell method), 299

magic_logstop() (IPython.core.magic.Magic method), 334

magic_magic() (IPython.core.interactiveshell.InteractiveShell method), 299

magic_magic() (IPython.core.magic.Magic method), 334

magic_macro() (IPython.core.interactiveshell.InteractiveShell method), 299

magic_macro() (IPython.core.magic.Magic method), 334

magic_magic() (IPython.core.interactiveshell.InteractiveShell method), 300

magic_magic() (IPython.core.magic.Magic method), 335

magic_matches() (IPython.core.completer.IPCompleter method), 225

magic_page() (IPython.core.interactiveshell.InteractiveShell method), 300

magic_page() (IPython.core.magic.Magic method), 335

magic_pdb() (IPython.core.interactiveshell.InteractiveShell method), 300

magic_pdb() (IPython.core.magic.Magic method), 336

magic_pdef() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_pdef() (IPython.core.magic.Magic method), 336

magic_pdoc() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_pdoc() (IPython.core.magic.Magic method), 336

magic_pfile() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_pfile() (IPython.core.magic.Magic method), 336

magic_pinfol() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_pinfol() (IPython.core.magic.Magic method), 336

magic_pinfol2() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_pinfol2() (IPython.core.magic.Magic method), 336

magic_popd() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_popd() (IPython.core.magic.Magic method), 336

magic_pprint() (IPython.core.interactiveshell.InteractiveShell method), 301

method), 301

magic_pprint() (IPython.core.magic.Magic method), 336

magic_profile() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_profile() (IPython.core.magic.Magic method), 336

magic_prun() (IPython.core.interactiveshell.InteractiveShell method), 301

magic_prun() (IPython.core.magic.Magic method), 336

magic_psearch() (IPython.core.interactiveshell.InteractiveShell method), 303

magic_psearch() (IPython.core.magic.Magic method), 338

magic_psources() (IPython.core.interactiveshell.InteractiveShell method), 304

magic_psources() (IPython.core.magic.Magic method), 339

magic_pushd() (IPython.core.interactiveshell.InteractiveShell method), 304

magic_pushd() (IPython.core.magic.Magic method), 339

magic_pwd() (IPython.core.interactiveshell.InteractiveShell method), 304

magic_pwd() (IPython.core.magic.Magic method), 339

magic_pwd() (IPython.kernel.core.magic.Magic method), 445

magic_pycat() (IPython.core.interactiveshell.InteractiveShell method), 304

magic_pycat() (IPython.core.magic.Magic method), 339

magic_pylab() (IPython.core.interactiveshell.InteractiveShell method), 304

magic_pylab() (IPython.core.magic.Magic method), 339

magic_quickref() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_quickref() (IPython.core.magic.Magic method), 340

magic_Quit() (IPython.core.interactiveshell.InteractiveShell method), 292

magic_quit() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_Quit() (IPython.core.magic.Magic method), 327

magic_quit() (IPython.core.magic.Magic method), 340

magic_r() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_r() (IPython.core.magic.Magic method), 340

magic_rehashx() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_rehashx() (IPython.core.magic.Magic method), 340

magic_reload_ext() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_reload_ext() (IPython.core.magic.Magic method), 340

magic_reset() (IPython.core.interactiveshell.InteractiveShell method), 305

magic_reset() (IPython.core.magic.Magic method), 340

magic_reset_selective() (IPython.core.interactiveshell.InteractiveShell method), 306

magic_reset_selective() (IPython.core.magic.Magic method), 341

magic_run() (IPython.core.interactiveshell.InteractiveShell method), 306

magic_run() (IPython.core.magic.Magic method), 341

magic_run_completer() (in module IPython.core.completerlib), 226

magic_save() (IPython.core.interactiveshell.InteractiveShell method), 308

magic_save() (IPython.core.magic.Magic method), 343

magic_sc() (IPython.core.interactiveshell.InteractiveShell method), 308

magic_sc() (IPython.core.magic.Magic method), 343

magic_sx() (IPython.core.interactiveshell.InteractiveShell method), 310

magic_shell() (IPython.core.magic.Magic method), 345

magic_tb() (IPython.core.interactiveshell.InteractiveShell method), 310

magic_tb() (IPython.core.magic.Magic method), 345

magic_time() (IPython.core.interactiveshell.InteractiveShell method), 310

magic_time() (IPython.core.magic.Magic method), 345

magic_timeit() (IPython.core.interactiveshell.InteractiveShell

method), 311

magic_timeit() (IPython.core.magic.Magic method), 346

magic_unalias() (IPython.core.interactiveshell.InteractiveShell method), 311

magic_unalias() (IPython.core.magic.Magic method), 346

magic_unload_ext() (IPython.core.interactiveshell.InteractiveShell method), 311

magic_unload_ext() (IPython.core.magic.Magic method), 346

magic_who() (IPython.core.interactiveshell.InteractiveShell method), 312

magic_who() (IPython.core.magic.Magic method), 347

magic_who_ls() (IPython.core.interactiveshell.InteractiveShell method), 312

magic_who_ls() (IPython.core.magic.Magic method), 347

magic_whos() (IPython.core.interactiveshell.InteractiveShell method), 312

magic_whos() (IPython.core.magic.Magic method), 347

magic_xmode() (IPython.core.interactiveshell.InteractiveShell method), 312

magic_xmode() (IPython.core.magic.Magic method), 347

MagicHandler (class in IPython.core.prefilter), 380

main() (in module IPython.lib.irunner), 618

main() (in module IPython.testing.ipctest), 627

main() (in module IPython.testing.mkdoctests), 631

main() (in module IPython.utils.pickleshare), 668

main() (in module IPython.utils.PyColorize), 641

main() (IPython.lib.irunner.InteractiveRunner method), 615

main() (IPython.lib.irunner.IPythonRunner method), 614

main() (IPython.lib.irunner.PythonRunner method), 616

main() (IPython.lib.irunner.SAGERunner method), 617

make_color_table() (in module IPython.utils.coloransi), 648

make_deferred() (in module IPython.kernel.twistedutil), 583

make_exclude() (in module IPython.testing.ipctest), 627

make_label_dec() (in module IPython.testing.decorators), 622

make_quoted_expr() (in module IPython.kernel.core.util), 458

make_quoted_expr() (in module IPython.utils.text), 681

make_report() (IPython.core.crashhandler.CrashHandler method), 228

make_runners() (in module IPython.testing.ipctest), 627

make_user_namespaces() (IPython.core.interactiveshell.InteractiveShell method), 312

Map (class in IPython.kernel.map), 506

map() (IPython.kernel.mapper.MultiEngineMapper method), 514

map() (IPython.kernel.mapper.SynchronousTaskMapper method), 514

map() (IPython.kernel.mapper.TaskMapper method), 515

map() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 546

map() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), 557

map_shell_method() (in module IPython.utils.attic), 642

mapper() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 546

mapper() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), 557

mark_dirs() (in module IPython.core.completer), 226

marquee() (in module IPython.utils.text), 681

marquee() (IPython.lib.demo.ClearDemo method), 594

marquee() (IPython.lib.demo.ClearIPDemo method), 596

marquee() (IPython.lib.demo.ClearMixin method), 597

marquee() (IPython.lib.demo.Demo method), 598

marquee() (IPython.lib.demo.IPythonDemo method), 600

marquee() (IPython.lib.demo.IPythonLineDemo method), 601

marquee() (IPython.lib.demo.LineDemo method), 603

master_config (IPython.core.application.Application attribute), 215

max_width (IPython.core.formatters.PlainTextFormatter

- attribute), 267
- merge() (IPython.utils.ipstruct.Struct method), 658
- merge_configs() (IPython.core.application.Application method), 215
- merge_configs() (IPython.kernel.ipclusterapp.IPClusterApp method), 499
- merge_configs() (IPython.kernel.ipengineapp.IPEngineApp method), 503
- message (IPython.config.loader.ConfigError attribute), 207
- message (IPython.config.loader.ConfigLoaderError attribute), 208
- message (IPython.core.alias.AliasError attribute), 210
- message (IPython.core.alias.InvalidAliasError attribute), 212
- message (IPython.core.application.ApplicationError attribute), 216
- message (IPython.core.error.IPythonCoreError attribute), 247
- message (IPython.core.error.TryNext attribute), 247
- message (IPython.core.error.UsageError attribute), 248
- message (IPython.core.interactiveshell.MultipleInstanceError attribute), 322
- message (IPython.core.interactiveshell.SpaceInInput attribute), 323
- message (IPython.core.prefilter.PrefilterError attribute), 384
- message (IPython.kernel.clientconnector.ClientConnectorError attribute), 417
- message (IPython.kernel.clientconnector.ClusterStateError attribute), 420
- message (IPython.kernel.core.error.ControllerCreationError attribute), 435
- message (IPython.kernel.core.error.ControllerError attribute), 435
- message (IPython.kernel.core.error.EngineCreationError attribute), 435
- message (IPython.kernel.core.error.EngineError attribute), 435
- message (IPython.kernel.core.error.IPythonError attribute), 436
- message (IPython.kernel.engineconnector.EngineConnectorError attribute), 460
- message (IPython.kernel.error.AbortedPendingDeferredTaskError attribute), 491
- message (IPython.kernel.error.ClientError attribute), 491
- message (IPython.kernel.error.CompositeError attribute), 491
- message (IPython.kernel.error.ConnectionError attribute), 491
- message (IPython.kernel.error.FileTimeoutError attribute), 492
- message (IPython.kernel.error.IdInUse attribute), 492
- message (IPython.kernel.error.InvalidClientID attribute), 492
- message (IPython.kernel.error.InvalidDeferredID attribute), 492
- message (IPython.kernel.error.InvalidEngineID attribute), 493
- message (IPython.kernel.error.InvalidProperty attribute), 493
- message (IPython.kernel.error.KernelError attribute), 493
- message (IPython.kernel.error.MessageSizeError attribute), 493
- message (IPython.kernel.error.MissingBlockArgument attribute), 493
- message (IPython.kernel.error.NoEnginesRegistered attribute), 494
- message (IPython.kernel.error.NotAPendingResult attribute), 494
- message (IPython.kernel.error.NotDefined attribute), 494
- message (IPython.kernel.error.PBMessageSizeError attribute), 494
- message (IPython.kernel.error.ProtocolError attribute), 495
- message (IPython.kernel.error.QueueCleared attribute), 495
- message (IPython.kernel.error.ResultAlreadyRetrieved attribute), 495
- message (IPython.kernel.error.ResultNotCompleted attribute), 495
- message (IPython.kernel.error.SecurityError attribute), 496
- message (IPython.kernel.error.SerializationError attribute), 496
- message (IPython.kernel.error.StopLocalExecution attribute), 496
- message (IPython.kernel.error.TaskAborted attribute), 496
- message (IPython.kernel.error.TaskRejectError attribute), 496

tribute), 497

message (IPython.kernel.error.TaskTimeout attribute), 497

message (IPython.kernel.error.UnpickleableException attribute), 497

message (IPython.lib.demos.DemoError attribute), 599

message (IPython.utils.growl.IPythonGrowlError attribute), 653

message (IPython.utils.notification.NotificationError attribute), 663

message (IPython.utils.path.HomeDirError attribute), 664

message (IPython.utils.process.FindCmdError attribute), 669

MessageSizeError (class in IPython.kernel.error), 493

MissingBlockArgument (class in IPython.kernel.error), 493

mktempfile() (IPython.core.interactiveshell.InteractiveShell method), 313

mktmp() (IPython.testing.tools.TempFileMixin method), 638

module_completer() (in module IPython.core.completerlib), 226

module_completion() (in module IPython.core.completerlib), 226

module_list() (in module IPython.core.completerlib), 227

mpl_runner() (in module IPython.lib.pylabtools), 620

multi_line_specials (IPython.core.prefilter.PrefilterManager attribute), 387

MultiEngine (class in IPython.kernel.multiengine), 540

MultiEngineMapper (class in IPython.kernel.mapper), 514

MultiLineMagicChecker (class in IPython.core.prefilter), 382

multiple_replace() (in module IPython.core.prompts), 399

multiple_replace() (in module IPython.kernel.core.prompts), 452

MultipleInstanceError (class in IPython.core.interactiveshell), 322

mutex_opts() (in module IPython.utils.attic), 642

myStringIO (class in IPython.core.oinspect), 350

N

n (IPython.utils.text.LSString attribute), 677

n (IPython.utils.text.SList attribute), 680

name (IPython.core.history.HistorySaveThread attribute), 273

name (IPython.kernel.twistedutil.ReactorInThread attribute), 583

name (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 586

name (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 586

name (IPython.lib.backgroundjobs.BackgroundJobFunc attribute), 587

names() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method), 422

names() (IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method), 424

names() (IPython.kernel.controllerservice.IControllerBase class method), 429

names() (IPython.kernel.controllerservice.IControllerCore class method), 431

names() (IPython.kernel.enginefc.IFCControllerBase class method), 465

names() (IPython.kernel.enginefc.IFCEngine class method), 467

names() (IPython.kernel.engineservice.IEngineBase class method), 472

names() (IPython.kernel.engineservice.IEngineCore class method), 475

names() (IPython.kernel.engineservice.IEngineProperties class method), 477

names() (IPython.kernel.engineservice.IEngineQueued class method), 479

names() (IPython.kernel.engineservice.IEngineSerialized class method), 482

names() (IPython.kernel.engineservice.IEngineThreaded class method), 484

names() (IPython.kernel.mapper.IMapper class method), 508

names() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 511

names() (IPython.kernel.mapper.ITaskMapperFactory class method), 513

names() (IPython.kernel.multiengine.IEngineMultiplexer class method), 518

names() (IPython.kernel.multiengine.IFullMultiEngine class method), 520

names() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 523	names() (IPython.kernel.enginefc.IFCEngine class method), 467
names() (IPython.kernel.multiengine.IMultiEngine class method), 525	namesAndDescriptions() (IPython.kernel.engineservice.IEngineBase class method), 472
names() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527	namesAndDescriptions() (IPython.kernel.engineservice.IEngineCore class method), 475
names() (IPython.kernel.multiengine.IMultiEngineExtras class method), 530	namesAndDescriptions() (IPython.kernel.engineservice.IEngineProperties class method), 477
names() (IPython.kernel.multiengine.ISynchronousEngine class method), 532	namesAndDescriptions() (IPython.kernel.engineservice.IEngineQueued class method), 479
names() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 535	namesAndDescriptions() (IPython.kernel.engineservice.IEngineSerialized class method), 482
names() (IPython.kernel.multiengine.ISynchronousMultiEngineBlocking class method), 537	namesAndDescriptions() (IPython.kernel.engineservice.IEngineThreaded class method), 484
names() (IPython.kernel.multiengine.ISynchronousMultiEngineNonBlocking class method), 539	namesAndDescriptions() (IPython.kernel.mapper.IMapper class method), 508
names() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 550	namesAndDescriptions() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 511
names() (IPython.kernel.multiengineclient.IPendingResult class method), 553	namesAndDescriptions() (IPython.kernel.mapper.ITaskMapperFactory class method), 513
names() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 562	namesAndDescriptions() (IPython.kernel.multiengine.IEngineMultiplexer class method), 518
names() (IPython.kernel.newserialized.ISerialized class method), 565	namesAndDescriptions() (IPython.kernel.multiengine.IFullMultiEngine class method), 520
names() (IPython.kernel.newserialized.IUnSerialized class method), 567	namesAndDescriptions() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 523
names() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator class method), 571	namesAndDescriptions() (IPython.kernel.multiengine.IMultiEngine class method), 525
names() (IPython.kernel.parallelfunction.IParallelFunction class method), 573	namesAndDescriptions() (IPython.kernel.multiengine.IMultiEngineCoordinator class method), 527
names() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 576	namesAndDescriptions() (IPython.kernel.multiengine.IMultiEngineExtras class method), 530
namesAndDescriptions() (IPython.kernel.clientinterfaces.IBlockingClientAdapter class method), 422	namesAndDescriptions()
namesAndDescriptions() (IPython.kernel.clientinterfaces.IFCCClientInterfaceProvider class method), 424	namesAndDescriptions()
namesAndDescriptions() (IPython.kernel.controllerservice.IControllerBase class method), 429	namesAndDescriptions()
namesAndDescriptions() (IPython.kernel.controllerservice.IControllerCore class method), 431	namesAndDescriptions()
namesAndDescriptions() (IPython.kernel.enginefc.IFCCControllerBase class method), 465	namesAndDescriptions()
namesAndDescriptions()	namesAndDescriptions()

(IPython.kernel.multiengine.ISynchronousEngineMultiplexor(IPython.core.debugger.Pdb method), class method), 532

namesAndDescriptions() new_main_mod() (IPython.core.interactiveshell.InteractiveShell (IPython.kernel.multiengine.ISynchronousMultiEngine method), 313 class method), 535

namesAndDescriptions() newline (IPython.core.formatters.PlainTextFormatter attribute), 267

(IPython.kernel.multiengine.ISynchronousMultiEngine(IPython.core.oinspect.myStringIO method), class method), 537

namesAndDescriptions() NextHandler (IPython.lib.inputhookwx.EventLoopTimer (IPython.kernel.multiengine.ISynchronousMultiEngine attribute), 611 class method), 539

namesAndDescriptions() NLprinter (class in IPython.utils.io), 654

(IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient.SList attribute), 680

class method), 550

namesAndDescriptions() no_op() (in module IPython.core.interactiveshell), 323

(IPython.kernel.multiengineclient.IPendingResult) (class in class method), 553

namesAndDescriptions() IPython.kernel.error), 494

(IPython.kernel.multiengineclient.IFCSynchronousMultiEngine noinfo() (IPython.core.oinspect.Inspector method), class method), 562

namesAndDescriptions() NotAPendingResult (class in IPython.kernel.error), 494

(IPython.kernel.newserialized.ISerialized NotDefined (class in class method), 565

namesAndDescriptions() IPython.kernel.core.interpreter), 444

(IPython.kernel.newserialized.IUnSerialized NotDefined (class in IPython.kernel.error), 494 class method), 567

namesAndDescriptions() NotGiven (class in IPython.utils.attic), 642

(IPython.kernel.parallelfunction.IMultiEngineParallelDecorator (class in IPython.utils.notification), 662 class method), 571

namesAndDescriptions() IPython.utils.notification), 663

(IPython.kernel.parallelfunction.IParallelFunction) (class in IPython.utils.growl), 653

class method), 573

namesAndDescriptions() notify() (in module IPython.utils.growl), 653

(IPython.kernel.parallelfunction.ITaskParallelDecorator (IPython.lib.inputhookwx.EventLoopTimer method), 611 class method), 576

namesAndDescriptions() IPython.utils.growl.Notifier method), 653

notify_deferred() (in module IPython.utils.growl), 653

NameSpace (class in IPython.utils.wildcard), 684

native_line_ends() (in module IPython.utils.text), notify_deferred() (IPython.utils.growl.Notifier method), 653

new() (IPython.lib.backgroundjobs.BackgroundJobManager(IPython.utils.wildcard.NameSpace attribute), method), 588

new_do_down() (IPython.core.debugger.Pdb ns_names (IPython.utils.wildcard.NameSpace attribute), 685 method), 235

new_do_frame() (IPython.core.debugger.Pdb num_cpus() (in module IPython.utils.sysinfo), 671 method), 235

new_do_quit() (IPython.core.debugger.Pdb num_ini_spaces() (in module IPython.core.inputsplitters), 282 method), 235

new_do_restart() (IPython.core.debugger.Pdb num_ini_spaces() (in module IPython.utils.text), 682 method), 235

numpy_not_available() (in module IPython.testing.decorators), 622

numpy_not_available() (in module IPython.testing.decorators_trial), 623

O

object_find() (IPython.kernel.core.magic.Magic method), 445

object_info() (in module IPython.core.oinspect), 353

object_info_string_level (IPython.core.interactiveshell.InteractiveShell attribute), 313

object_inspect() (IPython.core.interactiveshell.InteractiveShell method), 313

ofind() (IPython.core.prefilter.LineInfo method), 380

on_err_write() (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), 452

on_n_engines_registered_do() (IPython.kernel.controllerservice.ControllerAdapterBase method), 426

on_n_engines_registered_do() (IPython.kernel.controllerservice.ControllerService method), 427

on_n_engines_registered_do() (IPython.kernel.multiengine.MultiEngine method), 541

on_off() (in module IPython.core.magic), 348

on_out_write() (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), 453

on_register_engine_do() (IPython.kernel.controllerservice.ControllerAdapterBase method), 426

on_register_engine_do() (IPython.kernel.controllerservice.ControllerService method), 427

on_register_engine_do() (IPython.kernel.multiengine.MultiEngine method), 541

on_register_engine_do_not() (IPython.kernel.controllerservice.ControllerAdapterBase method), 426

on_register_engine_do_not() (IPython.kernel.controllerservice.ControllerService method), 427

on_register_engine_do_not() (IPython.kernel.multiengine.MultiEngine method), 541

on_trait_change() (IPython.core.alias.AliasManager method), 210

on_trait_change() (IPython.core.builtin_trap.BuiltinTrap method), 218

on_trait_change() (IPython.core.display_trap.DisplayTrap method), 239

on_trait_change() (IPython.core.displayhook.DisplayHook method), 242

on_trait_change() (IPython.core.displaypub.DisplayPublisher method), 244

on_trait_change() (IPython.core.extensions.ExtensionManager method), 249

on_trait_change() (IPython.core.formatters.BaseFormatter method), 253

on_trait_change() (IPython.core.formatters.DisplayFormatter method), 255

on_trait_change() (IPython.core.formatters.HTMLFormatter method), 257

on_trait_change() (IPython.core.formatters.JSONFormatter method), 260

on_trait_change() (IPython.core.formatters.LatexFormatter method), 262

on_trait_change() (IPython.core.formatters.PlainTextFormatter method), 267

on_trait_change() (IPython.core.formatters.PNGFormatter method), 264

on_trait_change() (IPython.core.formatters.SVGFormatter method), 266

on_trait_change() (IPython.core.interactiveshell.InteractiveShell method), 313

on_trait_change() (IPython.core.payload.PayloadManager method), 355

on_trait_change() (IPython.core.plugin.Plugin method), 358

on_trait_change() (IPython.core.plugin.PluginManager method), 359

on_trait_change() (IPython.core.prefilter.AliasChecker method), 362

on_trait_change() (IPython.core.prefilter.AliasHandler method), 363

on_trait_change() (IPython.core.prefilter.AssignMagicTransformer method), 364

on_trait_change() (IPython.core.prefilter.AssignmentChecker method), 367

on_trait_change() (IPython.core.prefilter.AssignSystemTransformer method), 366

on_trait_change() (IPython.core.prefilter.AutocallChecker method), 371

on_trait_change() (IPython.core.prefilter.AutoHandler method), 368

[on_trait_change\(\) \(IPython.core.prefilter.AutoMagicChecker method\), 427](#)
[method\), 370](#)
[on_trait_change\(\) \(IPython.core.prefilter.EmacsChecker \(IPython.kernel.multiengine.MultiEngine method\), 372](#)
[method\), 372](#)
[on_trait_change\(\) \(IPython.core.prefilter.EmacsHandler.oncmd\(\) \(IPython.core.debugger.Pdb method\), 235](#)
[method\), 374](#)
[method\), 374](#)
[on_trait_change\(\) \(IPython.core.prefilter.EscCharsChecker 644](#)
[method\), 375](#)
[method\), 375](#)
[on_trait_change\(\) \(IPython.core.prefilter.HelpHandler ostream \(IPython.core.ultratb.AutoFormattedTB at-](#)
[method\), 376](#)
[tribute\), 401](#)
[on_trait_change\(\) \(IPython.core.prefilter.IPyAutocallChecker \(IPython.core.ultratb.ColorTB attribute\),](#)
[method\), 377](#)
[403](#)
[on_trait_change\(\) \(IPython.core.prefilter.IPyPromptTransformer \(IPython.core.ultratb.FormattedTB at-](#)
[method\), 379](#)
[tribute\), 405](#)
[on_trait_change\(\) \(IPython.core.prefilter.MagicHandler ostream \(IPython.core.ultratb.ListTB attribute\), 406](#)
[method\), 381](#)
[ostream \(IPython.core.ultratb.SyntaxTB attribute\),](#)
[on_trait_change\(\) \(IPython.core.prefilter.MultiLineMagicChecker 407](#)
[method\), 382](#)
[ostream \(IPython.core.ultratb.TBTools attribute\),](#)
[on_trait_change\(\) \(IPython.core.prefilter.PrefilterChecker 408](#)
[method\), 383](#)
[ostream \(IPython.core.ultratb.VerboseTB attribute\),](#)
[on_trait_change\(\) \(IPython.core.prefilter.PrefilterHandler 410](#)
[method\), 385](#)
[osx_clipboard_get\(\) \(in module](#)
[on_trait_change\(\) \(IPython.core.prefilter.PrefilterManager IPython.lib.clipboard\), 589](#)
[method\), 387](#)
[out_text \(IPython.kernel.core.output_trap.OutputTrap](#)
[on_trait_change\(\) \(IPython.core.prefilter.PrefilterTransformer attribute\), 448](#)
[method\), 389](#)
[out_text \(IPython.kernel.core.redirector_output_trap.RedirectorOutput](#)
[on_trait_change\(\) \(IPython.core.prefilter.PyPromptTransformer attribute\), 453](#)
[method\), 390](#)
[OutputTrap \(class in](#)
[on_trait_change\(\) \(IPython.core.prefilter.PythonOpsChecker IPython.kernel.core.output_trap\), 447](#)
[method\), 392](#)
[Owner \(IPython.lib.inputhookwx.EventLoopTimer](#)
[on_trait_change\(\) \(IPython.core.prefilter.ShellEscapeChecker attribute\), 611](#)
[method\), 393](#)
P
[on_trait_change\(\) \(IPython.core.prefilter.ShellEscapeHandler](#)
[method\), 394](#)
[p \(IPython.utils.text.LSString attribute\), 677](#)
[on_unregister_engine_do\(\) p \(IPython.utils.text.SList attribute\), 680](#)
[\(IPython.kernel.controllerservice.ControllerAdapterBase \(IPython.core.prompts.BasePrompt at-](#)
[method\), 426](#)
[tribute\), 396](#)
[on_unregister_engine_do\(\) p_template \(IPython.core.prompts.Prompt1 at-](#)
[\(IPython.kernel.controllerservice.ControllerService tribute\), 397](#)
[method\), 427](#)
[p_template \(IPython.core.prompts.Prompt2 at-](#)
[on_unregister_engine_do\(\) p_template \(IPython.core.prompts.PromptOut at-](#)
[\(IPython.kernel.multiengine.MultiEngine p_template \(IPython.core.prompts.PromptOut at-](#)
[method\), 541](#)
[tribute\), 398](#)
[tribute\), 398](#)
[on_unregister_engine_do_not\(\) p_template \(IPython.kernel.core.prompts.BasePrompt](#)
[\(IPython.kernel.controllerservice.ControllerAdapterBase attribute\), 449](#)
[method\), 426](#)
[p_template \(IPython.kernel.core.prompts.Prompt1](#)
[on_unregister_engine_do_not\(\) attribute\), 450](#)
[\(IPython.kernel.controllerservice.ControllerService](#)

- p_template (IPython.kernel.core.prompts.Prompt2 attribute), 451
- p_template (IPython.kernel.core.prompts.PromptOut attribute), 451
- pack_exception() (IPython.kernel.core.interpreter.Interpreter method), 443
- packageFailure() (in module IPython.kernel.pbutil), 577
- packageFailure() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559
- packageResult() (in module IPython.kernel.multienginefc), 562
- packageSuccess() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559
- page() (in module IPython.core.page), 354
- page() (in module IPython.core.payloadpage), 357
- page_dumb() (in module IPython.core.page), 354
- page_file() (in module IPython.core.page), 354
- parallel() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 546
- parallel() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient method), 557
- ParallelFunction (class in IPython.kernel.parallelfunction), 576
- Parametric() (in module IPython.testing.parametric), 632
- parametric() (in module IPython.testing.parametric), 632
- params (IPython.testing.ipctest.IPTester attribute), 627
- parse_args() (IPython.config.loader.ArgumentParser method), 205
- parse_known_args() (IPython.config.loader.ArgumentParser method), 205
- parse_options() (IPython.core.interactiveshell.InteractiveShell method), 314
- parse_options() (IPython.core.magic.Magic method), 347
- parse_test_output() (in module IPython.testing.tools), 639
- parseline() (IPython.core.debugger.Pdb method), 235
- Parser (class in IPython.utils.PyColorize), 640
- parseResults() (in module IPython.kernel.twistedutil), 583
- partial() (in module IPython.testing.parametric), 632
- partition (IPython.utils.text.LSString attribute), 677
- PATH, 2
- paths (IPython.utils.text.LSString attribute), 677
- paths (IPython.utils.text.SList attribute), 680
- pause() (IPython.kernel.twistedutil.DeferredList method), 582
- payload_manager (IPython.core.interactiveshell.InteractiveShell attribute), 314
- PayloadManager (class in IPython.core.payload), 314
- PBMessageSizeError (class in IPython.kernel.error), 494
- Pdb (class in IPython.core.debugger), 229
- pdb (IPython.core.interactiveshell.InteractiveShell attribute), 314
- pdef() (IPython.core.oinspect.Inspector method), 349
- pdoc() (IPython.core.oinspect.Inspector method), 350
- PendingDeferredManager (class in IPython.kernel.pendingdeferred), 578
- PendingDeferredManager (class in IPython.kernel.multiengineclient), 554
- pexpect_monkeypatch() (in module IPython.lib.irunner), 618
- pfile() (IPython.core.oinspect.Inspector method), 350
- PickleShareDB (class in IPython.utils.pickleshare), 666
- PickleShareLink (class in IPython.utils.pickleshare), 668
- pids (IPython.testing.ipctest.IPTester attribute), 627
- pinfo() (IPython.core.oinspect.Inspector method), 350
- pkg_commit_hash() (in module IPython.utils.sysinfo), 671
- plain() (in module IPython.utils.sysinfo), 671
- plain() (IPython.core.ultratb.AutoFormattedTB method), 402
- plain() (IPython.core.ultratb.ColorTB method), 403
- plain() (IPython.core.ultratb.FormattedTB method), 405
- plain_text_only (IPython.core.formatters.DisplayFormatter attribute), 255
- PlainTextFormatter (class in IPython.core.formatters), 265
- PlainTracebackFormatter (class in IPython.kernel.core.traceback_formatter), 454

- Plugin (class in IPython.core.plugin), 358
- plugin_manager (IPython.core.interactiveshell.InteractiveShell attribute), 314
- PluginManager (class in IPython.core.plugin), 359
- plugins (IPython.core.plugin.PluginManager attribute), 359
- PNGFormatter (class in IPython.core.formatters), 263
- pop (IPython.config.loader.Config attribute), 206
- pop (IPython.kernel.core.util.Bunch attribute), 457
- pop (IPython.kernel.core.util.InputList attribute), 458
- pop (IPython.kernel.multiengineclient.QueueStatusList attribute), 555
- pop (IPython.kernel.multiengineclient.ResultList attribute), 556
- pop (IPython.testing.globalipapp.ipnsdict attribute), 625
- pop (IPython.utils.coloransi.ColorSchemeTable attribute), 647
- pop (IPython.utils.ipstruct.Struct attribute), 660
- pop (IPython.utils.text.SList attribute), 680
- pop() (IPython.kernel.engineservice.StrictDict method), 487
- pop() (IPython.utils.pickleshare.PickleShareDB method), 667
- popitem (IPython.config.loader.Config attribute), 206
- popitem (IPython.kernel.core.util.Bunch attribute), 457
- popitem (IPython.testing.globalipapp.ipnsdict attribute), 625
- popitem (IPython.utils.coloransi.ColorSchemeTable attribute), 647
- popitem (IPython.utils.ipstruct.Struct attribute), 660
- popitem() (IPython.kernel.engineservice.StrictDict method), 487
- popitem() (IPython.utils.pickleshare.PickleShareDB method), 667
- popkey() (in module IPython.utils.attic), 642
- populate_readline_history() (IPython.core.history.HistoryManager method), 272
- post_cmd() (IPython.lib.demos.ClearDemo method), 594
- post_cmd() (IPython.lib.demos.ClearIPDemo method), 596
- post_cmd() (IPython.lib.demos.ClearMixin method), 597
- post_cmd() (IPython.lib.demos.Demo method), 598
- post_cmd() (IPython.lib.demos.IPythonDemo method), 600
- post_cmd() (IPython.lib.demos.IPythonLineDemo method), 602
- post_cmd() (IPython.lib.demos.LineDemo method), 603
- post_construct() (IPython.core.application.Application method), 215
- post_construct() (IPython.kernel.ipclusterapp.IPClusterApp method), 499
- post_construct() (IPython.kernel.ipengineapp.IPEngineApp method), 503
- post_load_command_line_config() (IPython.core.application.Application method), 215
- post_load_command_line_config() (IPython.kernel.ipclusterapp.IPClusterApp method), 499
- post_load_command_line_config() (IPython.kernel.ipengineapp.IPEngineApp method), 503
- post_load_file_config() (IPython.core.application.Application method), 215
- post_load_file_config() (IPython.kernel.ipclusterapp.IPClusterApp method), 499
- post_load_file_config() (IPython.kernel.ipengineapp.IPEngineApp method), 503
- post_notification() (IPython.utils.notification.NotificationCenter method), 662
- postcmd() (IPython.core.debugger.Pdb method), 235
- postloop() (IPython.core.debugger.Pdb method), 235
- pprint (IPython.core.formatters.PlainTextFormatter attribute), 267
- PPrintDisplayFormatter (class in IPython.kernel.core.display_formatter), 433
- pre_cmd() (IPython.lib.demos.ClearDemo method), 594
- pre_cmd() (IPython.lib.demos.ClearIPDemo method), 596
- pre_cmd() (IPython.lib.demos.ClearMixin method), 597
- pre_cmd() (IPython.lib.demos.Demo method), 598
- pre_cmd() (IPython.lib.demos.IPythonDemo method), 600

pre_cmd() (IPython.lib.demos.IPythonLineDemo method), 602

pre_cmd() (IPython.lib.demos.LineDemo method), 603

pre_construct() (IPython.core.application.Application method), 215

pre_construct() (IPython.kernel.ipclusterapp.IPClusterApp method), 499

pre_construct() (IPython.kernel.ipengineapp.IPEngineApp method), 504

pre_load_command_line_config() (IPython.core.application.Application method), 215

pre_load_command_line_config() (IPython.kernel.ipclusterapp.IPClusterApp method), 499

pre_load_command_line_config() (IPython.kernel.ipengineapp.IPEngineApp method), 504

pre_load_file_config() (IPython.core.application.Application method), 215

pre_load_file_config() (IPython.kernel.ipclusterapp.IPClusterApp method), 499

pre_load_file_config() (IPython.kernel.ipengineapp.IPEngineApp method), 504

pre_prompt_hook() (in module IPython.core.hooks), 276

pre_readline() (IPython.core.interactiveshell.InteractiveShell method), 314

pre_run_code_hook() (in module IPython.core.hooks), 277

precmd() (IPython.core.debugger.Pdb method), 235

prefilter_line() (IPython.core.prefilter.PrefilterManager method), 387

prefilter_line_info() (IPython.core.prefilter.PrefilterManager method), 388

prefilter_lines() (IPython.core.prefilter.PrefilterManager method), 388

prefilter_manager (IPython.core.interactiveshell.InteractiveShell attribute), 314

prefilter_manager (IPython.core.prefilter.AliasChecker attribute), 362

prefilter_manager (IPython.core.prefilter.AliasHandler attribute), 364

prefilter_manager (IPython.core.prefilter.AssignMagicTransformer attribute), 365

prefilter_manager (IPython.core.prefilter.AssignmentChecker attribute), 367

prefilter_manager (IPython.core.prefilter.AssignSystemTransformer attribute), 366

prefilter_manager (IPython.core.prefilter.AutocallChecker attribute), 371

prefilter_manager (IPython.core.prefilter.AutoHandler attribute), 369

prefilter_manager (IPython.core.prefilter.AutoMagicChecker attribute), 370

prefilter_manager (IPython.core.prefilter.EmacsChecker attribute), 373

prefilter_manager (IPython.core.prefilter.EmacsHandler attribute), 374

prefilter_manager (IPython.core.prefilter.EscCharsChecker attribute), 375

prefilter_manager (IPython.core.prefilter.HelpHandler attribute), 377

prefilter_manager (IPython.core.prefilter.IPyAutocallChecker attribute), 378

prefilter_manager (IPython.core.prefilter.IPyPromptTransformer attribute), 379

prefilter_manager (IPython.core.prefilter.MagicHandler attribute), 381

prefilter_manager (IPython.core.prefilter.MultiLineMagicChecker attribute), 382

prefilter_manager (IPython.core.prefilter.PrefilterChecker attribute), 384

prefilter_manager (IPython.core.prefilter.PrefilterHandler attribute), 385

prefilter_manager (IPython.core.prefilter.PrefilterTransformer attribute), 390

prefilter_manager (IPython.core.prefilter.PyPromptTransformer attribute), 391

prefilter_manager (IPython.core.prefilter.PythonOpsChecker attribute), 392

prefilter_manager (IPython.core.prefilter.ShellEscapeChecker attribute), 394

prefilter_manager (IPython.core.prefilter.ShellEscapeHandler attribute), 395

PrefilterChecker (class in IPython.core.prefilter), 383

PrefilterError (class in IPython.core.prefilter), 384

PrefilterHandler (class in IPython.core.prefilter), 385

PrefilterManager (class in IPython.core.prefilter), 386

PrefilterTransformer (class in IPython.core.prefilter), 387

- 389
- preloop() (IPython.core.debugger.Pdb method), 235
- prepending_to_syspath (class in IPython.utils.syspathcontext), 672
- PreviousHandler (IPython.lib.inputhookwx.EventLoopTimer attribute), 611
- print_help() (IPython.config.loader.ArgumentParser method), 205
- print_ipcluster_logs() (IPython.kernel.clientconnector.Cluster method), 419
- print_ipcontroller_logs() (IPython.kernel.clientconnector.Cluster method), 419
- print_ipengine_logs() (IPython.kernel.clientconnector.Cluster method), 419
- print_list_lines() (IPython.core.debugger.Pdb method), 235
- print_logs() (IPython.kernel.clientconnector.Cluster method), 419
- print_method (IPython.core.formatters.BaseFormatter attribute), 253
- print_method (IPython.core.formatters.HTMLFormatter attribute), 258
- print_method (IPython.core.formatters.JSONFormatter attribute), 260
- print_method (IPython.core.formatters.LatexFormatter attribute), 262
- print_method (IPython.core.formatters.PlainTextFormatter attribute), 267
- print_method (IPython.core.formatters.PNGFormatter attribute), 264
- print_method (IPython.core.formatters.SVGFormatter attribute), 269
- print_stack_entry() (IPython.core.debugger.Pdb method), 235
- print_stack_trace() (IPython.core.debugger.Pdb method), 235
- print_topics() (IPython.core.debugger.Pdb method), 235
- print_tracebacks() (IPython.kernel.error.CompositeError method), 491
- print_usage() (IPython.config.loader.ArgumentParser method), 205
- print_version() (IPython.config.loader.ArgumentParser method), 205
- printer() (in module IPython.kernel.util), 584
- priority (IPython.core.prefilter.AliasChecker attribute), 362
- priority (IPython.core.prefilter.AssignMagicTransformer attribute), 365
- priority (IPython.core.prefilter.AssignmentChecker attribute), 368
- priority (IPython.core.prefilter.AssignSystemTransformer attribute), 366
- priority (IPython.core.prefilter.AutocallChecker attribute), 371
- priority (IPython.core.prefilter.AutoMagicChecker attribute), 370
- priority (IPython.core.prefilter.EmacsChecker attribute), 373
- priority (IPython.core.prefilter.EscCharsChecker attribute), 375
- priority (IPython.core.prefilter.IPyAutocallChecker attribute), 378
- priority (IPython.core.prefilter.IPyPromptTransformer attribute), 379
- priority (IPython.core.prefilter.MultiLineMagicChecker attribute), 382
- priority (IPython.core.prefilter.PrefilterChecker attribute), 384
- priority (IPython.core.prefilter.PrefilterTransformer attribute), 390
- priority (IPython.core.prefilter.PyPromptTransformer attribute), 391
- priority (IPython.core.prefilter.PythonOpsChecker attribute), 392
- priority (IPython.core.prefilter.ShellEscapeChecker attribute), 394
- privileged_start_service() (IPython.kernel.controllerservice.ControllerService method), 427
- privileged_start_service() (IPython.kernel.engineservice.EngineService method), 470
- privileged_start_service() (IPython.kernel.engineservice.ThreadedEngineService method), 488
- ProcessEvent() (IPython.lib.inputhookwx.EventLoopTimer method), 611
- ProcessEventLocally() (IPython.lib.inputhookwx.EventLoopTimer method), 611
- ProcessPendingEvents() (IPython.lib.inputhookwx.EventLoopTimer method), 611

method), 611

processUniqueID() (IPython.kernel.enginefc.FCEngineReferenceFromShell static method), 465

processUniqueID() (IPython.kernel.enginefc.FCRemoteEngineReferenceFromShell static method), 467

processUniqueID() (IPython.kernel.multienginefc.FCSynchronousMultiEngineReferenceFromShell static method), 470

profile (IPython.core.interactiveshell.InteractiveShell attribute), 314

profile_missing_notice() (IPython.core.interactiveshell.InteractiveShell static method), 314

profile_missing_notice() (IPython.core.magic.Magic static method), 348

profile_name (IPython.core.application.Application attribute), 215

Prompt1 (class in IPython.core.prompts), 397

Prompt1 (class in IPython.kernel.core.prompts), 450

Prompt2 (class in IPython.core.prompts), 397

Prompt2 (class in IPython.kernel.core.prompts), 450

prompt_count (IPython.core.displayhook.DisplayHook attribute), 242

prompt_in1 (IPython.core.interactiveshell.InteractiveShell attribute), 314

prompt_in2 (IPython.core.interactiveshell.InteractiveShell attribute), 314

prompt_out (IPython.core.interactiveshell.InteractiveShell attribute), 315

PromptOut (class in IPython.core.prompts), 398

PromptOut (class in IPython.kernel.core.prompts), 451

prompts_pad_left (IPython.core.interactiveshell.InteractiveShell attribute), 315

properties (IPython.kernel.enginefc.EngineFromReference static method), 461

properties (IPython.kernel.engineservice.QueuedEngine static method), 485

protect_filename() (in module IPython.core.completer), 226

ProtocolError (class in IPython.kernel.error), 494

providedBy() (IPython.kernel.clientinterfaces.IBlockingClientAdapter static method), 422

providedBy() (IPython.kernel.clientinterfaces.IFCClientInterface static method), 424

providedBy() (IPython.kernel.controllerservice.IControllerBase static method), 429

providedBy() (IPython.kernel.controllerservice.IControllerCore static method), 432

providedBy() (IPython.kernel.enginefc.IFCControllerBase static method), 465

providedBy() (IPython.kernel.enginefc.IFCEngine static method), 467

providedBy() (IPython.kernel.engineservice.IEngineBase static method), 470

providedBy() (IPython.kernel.engineservice.IEngineCore static method), 475

providedBy() (IPython.kernel.engineservice.IEngineProperties static method), 477

providedBy() (IPython.kernel.engineservice.IEngineQueued static method), 480

providedBy() (IPython.kernel.engineservice.IEngineSerialized static method), 482

providedBy() (IPython.kernel.engineservice.IEngineThreaded static method), 484

providedBy() (IPython.kernel.mapper.IMapper static method), 509

providedBy() (IPython.kernel.mapper.IMultiEngineMapperFactory static method), 511

providedBy() (IPython.kernel.mapper.ITaskMapperFactory static method), 513

providedBy() (IPython.kernel.multiengine.IEngineMultiplexer static method), 518

providedBy() (IPython.kernel.multiengine.IFullMultiEngine static method), 520

providedBy() (IPython.kernel.multiengine.IFullSynchronousMultiEngine static method), 523

providedBy() (IPython.kernel.multiengine.IMultiEngine static method), 525

providedBy() (IPython.kernel.multiengine.IMultiEngineCoordinator static method), 528

providedBy() (IPython.kernel.multiengine.IMultiEngineExtras static method), 530

providedBy() (IPython.kernel.multiengine.ISynchronousEngineMulti static method), 532

providedBy() (IPython.kernel.multiengine.ISynchronousMultiEngine static method), 535

providedBy() (IPython.kernel.multiengine.ISynchronousMultiEngine static method), 537

providedBy() (IPython.kernel.multiengine.ISynchronousMultiEngine static method), 539

providedBy() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine static method), 550

providedBy() (IPython.kernel.multiengineclient.IPendingResult static method), 553

providedBy() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine static method), 562

providedBy() (IPython.kernel.newserialized.ISerializedPullFunction() (IPython.kernel.multiengine.SynchronousMultiEngineClient) static method), 565	pull_function() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) static method), 542
providedBy() (IPython.kernel.newserialized.IUnSerializedPullFunction() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) static method), 567	pull_function() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) static method), 547
providedBy() (IPython.kernel.parallelfunction.IMultiEngineParallelFunction() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient) static method), 571	pull_function() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient) static method), 558
providedBy() (IPython.kernel.parallelfunction.IParallelFunction() (IPython.kernel.enginefc.EngineFromReference) static method), 573	pull_function() (IPython.kernel.enginefc.EngineFromReference) static method), 461
providedBy() (IPython.kernel.parallelfunction.ITaskParallelFunction() (IPython.kernel.engineservice.EngineServiceClient) static method), 576	pull_function() (IPython.kernel.engineservice.EngineServiceClient) static method), 470
pssearch() (IPython.core.oinspect.Inspector method), 350	pull_serialized() (IPython.kernel.engineservice.QueuedEngineClient) method), 485
psource() (IPython.core.oinspect.Inspector method), 350	pull_serialized() (IPython.kernel.engineservice.ThreadedEngineServiceClient) method), 488
publish() (IPython.core.displaypub.DisplayPublisher method), 244	pull_serialized() (IPython.kernel.multiengine.MultiEngineClient) method), 541
publish_display_data() (in module IPython.core.displaypub), 245	pull_serialized() (IPython.kernel.multiengine.SynchronousMultiEngineClient) method), 542
pull() (IPython.kernel.core.interpreter.Interpreter method), 443	pull_serialized() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) method), 547
pull() (IPython.kernel.enginefc.EngineFromReference method), 461	pull_serialized() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient) method), 558
pull() (IPython.kernel.engineservice.EngineServiceClient) method), 470	push() (IPython.core.inputsplitters.InputSplitter method), 280
pull() (IPython.kernel.engineservice.QueuedEngineClient) method), 485	push() (IPython.core.inputsplitters.IPythonInputSplitter method), 278
pull() (IPython.kernel.engineservice.ThreadedEngineServiceClient) method), 488	push() (IPython.core.interactiveshell.InteractiveShell method), 315
pull() (IPython.kernel.multiengine.MultiEngineClient) method), 541	push() (IPython.kernel.core.interpreter.Interpreter method), 443
pull() (IPython.kernel.multiengine.SynchronousMultiEngineClient) method), 542	push() (IPython.kernel.enginefc.EngineFromReference) method), 461
pull() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) method), 546	push() (IPython.kernel.engineservice.EngineServiceClient) method), 470
pull() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient) method), 558	push() (IPython.kernel.engineservice.QueuedEngineClient) method), 486
pull_function() (IPython.kernel.core.interpreter.Interpreter method), 443	push() (IPython.kernel.engineservice.ThreadedEngineServiceClient) method), 488
pull_function() (IPython.kernel.enginefc.EngineFromReference) method), 461	push() (IPython.kernel.multiengine.MultiEngineClient) method), 541
pull_function() (IPython.kernel.engineservice.EngineServiceClient) method), 470	push() (IPython.kernel.multiengine.SynchronousMultiEngineClient) method), 542
pull_function() (IPython.kernel.engineservice.QueuedEngineClient) method), 485	push() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient) method), 547
pull_function() (IPython.kernel.engineservice.ThreadedEngineServiceClient) method), 488	push() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient) method), 558
pull_function() (IPython.kernel.multiengine.MultiEngineClient) method), 541	push_accepts_more() (IPython.core.inputsplitters.InputSplitter method), 280

method), 281

push_accepts_more() (IPython.core.inputsplitter.IPythonInputSplitter method), 279

push_function() (IPython.kernel.core.interpreter.Interpreter method), 443

push_function() (IPython.kernel.enginefc.EngineFromReference method), 461

push_function() (IPython.kernel.engineservice.EngineService method), 470

push_function() (IPython.kernel.engineservice.QueuedEngine method), 486

push_function() (IPython.kernel.engineservice.ThreadedEngine method), 488

push_function() (IPython.kernel.multiengine.MultiEngine method), 541

push_function() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542

push_function() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 547

push_function() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient class method), 558

push_line() (IPython.core.interactiveshell.InteractiveShell method), 315

push_serialized() (IPython.kernel.enginefc.EngineFromReference class method), 462

push_serialized() (IPython.kernel.engineservice.EngineService method), 470

push_serialized() (IPython.kernel.engineservice.QueuedEngine method), 486

push_serialized() (IPython.kernel.engineservice.ThreadedEngine method), 488

push_serialized() (IPython.kernel.multiengine.MultiEngine class method), 541

push_serialized() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542

push_serialized() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 548

push_serialized() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient class method), 558

py_file_finder (class in IPython.testing.globalipapp), 626

pycmd2argv() (in module IPython.utils.process), 669

PyFileConfigLoader (class in IPython.config.loader), 208

pyfunc() (in module IPython.testing.plugin.dtexample), 633

pyfunc() (in module IPython.testing.plugin.simple), 635

pylab_activate() (in module IPython.lib.pylabtools), 620

PyPromptTransformer (class in IPython.core.prefilter), 390

python_func_kw_matches() (IPython.core.completer.IPCCompleter method), 225

python_matches() (IPython.core.completer.IPCCompleter method), 225

PythonOpsChecker (class in IPython.core.prefilter), 392

PythonRefService (class in IPython.lib.irunner), 616

PYTHONSTARTUP, 693

Q

queryDescriptionFor() (IPython.kernel.clientinterfaces.IBlockingClientAdaptor method), 433

queryDescriptionFor() (IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method), 425

queryDescriptionFor() (IPython.kernel.controllerservice.IControllerBase class method), 429

queryDescriptionFor() (IPython.kernel.controllerservice.IControllerCore class method), 432

queryDescriptionFor() (IPython.kernel.enginefc.IFCControllerBase class method), 465

queryDescriptionFor() (IPython.kernel.enginefc.IFCEngine class method), 467

queryDescriptionFor() (IPython.kernel.engineservice.IEngineBase class method), 472

queryDescriptionFor() (IPython.kernel.engineservice.IEngineCore class method), 475

queryDescriptionFor() (IPython.kernel.engineservice.IEngineProperties class method), 477

queryDescriptionFor() (IPython.kernel.engineservice.IEngineQueued class method), 480

queryDescriptionFor() (IPython.kernel.engineservice.IEngineSerialized class method), 480

class method), 482

queryDescriptionFor()
(IPython.kernel.engineservice.IEngineThreaded
class method), 484

queryDescriptionFor()
(IPython.kernel.mapper.IMapper class
method), 509

queryDescriptionFor()
(IPython.kernel.mapper.IMultiEngineMapperFactory
class method), 511

queryDescriptionFor()
(IPython.kernel.mapper.ITaskMapperFactory
class method), 513

queryDescriptionFor()
(IPython.kernel.multiengine.IEngineMultiplexer
class method), 518

queryDescriptionFor()
(IPython.kernel.multiengine.IFullMultiEngine
class method), 521

queryDescriptionFor()
(IPython.kernel.multiengine.IFullSynchronousMultiEngine
class method), 523

queryDescriptionFor()
(IPython.kernel.multiengine.IMultiEngine
class method), 525

queryDescriptionFor()
(IPython.kernel.multiengine.IMultiEngineCoordinator
class method), 528

queryDescriptionFor()
(IPython.kernel.multiengine.IMultiEngineExtras
class method), 530

queryDescriptionFor()
(IPython.kernel.multiengine.ISynchronousEngineMultiEngine
class method), 532

queryDescriptionFor()
(IPython.kernel.multiengine.ISynchronousMultiEngine
class method), 535

queryDescriptionFor()
(IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator
class method), 537

queryDescriptionFor()
(IPython.kernel.multiengine.ISynchronousMultiEnginePython
class method), 540

queryDescriptionFor()
(IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient
class method), 551

queryDescriptionFor()
(IPython.kernel.multiengineclient.IPendingResult
class method), 553

queryDescriptionFor()
(IPython.kernel.multienginefc.IFCSynchronousMultiEngine
class method), 562

queryDescriptionFor()
(IPython.kernel.newserialized.ISerialized
class method), 565

queryDescriptionFor()
(IPython.kernel.newserialized.IUnSerialized
class method), 567

queryDescriptionFor()
(IPython.kernel.parallelfunction.IMultiEngineParallelDecorator
class method), 571

queryDescriptionFor()
(IPython.kernel.parallelfunction.IParallelFunction
class method), 573

queryDescriptionFor()
(IPython.kernel.parallelfunction.ITaskParallelDecorator
class method), 576

queryTaggedValue()
(IPython.kernel.clientinterfaces.IBlockingClientAdaptor
class method), 422

queryTaggedValue()
(IPython.kernel.clientinterfaces.IFCCClientInterfaceProvider
class method), 425

queryTaggedValue()
(IPython.kernel.controllerservice.IControllerBase
class method), 429

queryTaggedValue()
(IPython.kernel.controllerservice.IControllerCore
class method), 432

queryTaggedValue()
(IPython.kernel.enginefc.IFCCControllerBase
class method), 465

queryTaggedValue()
(IPython.kernel.enginefc.IFCEngine
class method), 467

queryTaggedValue()
(IPython.kernel.engineservice.IEngineBase
class method), 472

queryTaggedValue()
(IPython.kernel.engineservice.IEngineCore
class method), 475

queryTaggedValue()
(IPython.kernel.engineservice.IEngineProperties
class method), 477

queryTaggedValue()
(IPython.kernel.engineservice.IEngineQueued

class method), 480	class method), 551
queryTaggedValue() (IPython.kernel.engineservice.IEngineSerialized class method), 482	queryTaggedValue() (IPython.kernel.multiengineclient.IPendingResult class method), 553
queryTaggedValue() (IPython.kernel.engineservice.IEngineThreaded class method), 484	queryTaggedValue() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 562
queryTaggedValue() (IPython.kernel.mapper.IMapper class method), 509	queryTaggedValue() (IPython.kernel.newserialized.ISerialized class method), 565
queryTaggedValue() (IPython.kernel.mapper.IMultiEngineMapperFactory class method), 511	queryTaggedValue() (IPython.kernel.newserialized.IUnSerialized class method), 567
queryTaggedValue() (IPython.kernel.mapper.ITaskMapperFactory class method), 513	queryTaggedValue() (IPython.kernel.parallelfunction.IMultiEngineParallelDecor class method), 571
queryTaggedValue() (IPython.kernel.multiengine.IEngineMultiplexer class method), 518	queryTaggedValue() (IPython.kernel.parallelfunction.IParallelFunction class method), 574
queryTaggedValue() (IPython.kernel.multiengine.IFullMultiEngine class method), 521	queryTaggedValue() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 576
queryTaggedValue() (IPython.kernel.multiengine.IFullSynchronousMultiEngine class method), 523	queue() (in module IPython.kernel.engineservice), 480
queryTaggedValue() (IPython.kernel.multiengine.IMultiEngine class method), 525	queue_status() (IPython.kernel.engineservice.QueuedEngine method), 486
queryTaggedValue() (IPython.kernel.multiengine.IMultiEngineCoordinator method), class method), 528	queue_status() (IPython.kernel.multiengine.MultiEngine method), 541
queryTaggedValue() (IPython.kernel.multiengine.IMultiEngineExtensio class method), 530	queue_status() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousEng class method), 532	queue_status() (IPython.kernel.multiengineclient.FullBlockingMultiE method), 548
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	queue_status() (IPython.kernel.multienginefc.FCFullSynchronousMu method), 558
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	QueueCleared (class in IPython.kernel.error), 495
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	QueueEvent() (IPython.lib.inputhookwx.EventLoopTimer method), 611
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	QueueStatusList (class in IPython.kernel.multiengineclient), 555
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	quick_has_id() (IPython.core.completerlib), 227
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	quick_has_id() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542
queryTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngineCompleto class method), 537	quick_has_id() (IPython.kernel.pendingdeferred.PendingDeferredMa method), 578
queryTaggedValue() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient method), 537	InteractiveShell (IPython.interactiveshell.InteractiveShell class), 227

- attribute), 315
- quiet() (IPython.core.displayhook.DisplayHook method), 242
- qw() (in module IPython.utils.text), 682
- qw_lol() (in module IPython.utils.text), 682
- qwflat() (in module IPython.utils.text), 682
- R**
- r (IPython.kernel.multiengineclient.PendingResult attribute), 555
- raise_exception() (IPython.kernel.error.CompositeError method), 491
- random_all() (in module IPython.testing.plugin.dtexample), 633
- ranfunc() (in module IPython.testing.plugin.dtexample), 634
- raw_input_ext() (in module IPython.utils.io), 656
- raw_input_multi() (in module IPython.utils.io), 656
- raw_map() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 548
- raw_map() (IPython.kernel.multiengineclient.FCFullSynchronousMultiEngineClient method), 558
- raw_print() (in module IPython.utils.io), 656
- raw_print_err() (in module IPython.utils.io), 656
- re_mark() (in module IPython.lib.demo), 604
- ReactorInThread (class in IPython.kernel.twistedutil), 582
- read() (IPython.core.oinspect.myStringIO method), 351
- read_payload() (IPython.core.payload.PayloadManager method), 356
- readline() (IPython.core.oinspect.myStringIO method), 351
- readline_merge_completions (IPython.core.interactiveshell.InteractiveShell attribute), 315
- readline_omit__names (IPython.core.interactiveshell.InteractiveShell attribute), 315
- readline_parse_and_bind (IPython.core.interactiveshell.InteractiveShell attribute), 315
- readline_remove_delims (IPython.core.interactiveshell.InteractiveShell attribute), 315
- readline_use (IPython.core.interactiveshell.InteractiveShell attribute), 315
- readlines() (IPython.core.oinspect.myStringIO method), 351
- rebindFunctionGlobals() (in module IPython.kernel.pickleutil), 580
- RedirectorOutputTrap (class in IPython.kernel.core.redirector_output_trap), 452
- reduce_code() (in module IPython.kernel.codeutil), 425
- register() (IPython.config.loader.ArgumentParser method), 205
- register_checker() (IPython.core.prefilter.PrefilterManager method), 388
- register_engine() (IPython.kernel.controllerservice.ControllerAdapter method), 427
- register_engine() (IPython.kernel.controllerservice.ControllerService method), 427
- register_engine() (IPython.kernel.multiengine.MultiEngine method), 427
- register_engine_client (IPython.kernel.multiengineclient.MultiEngineClient method), 427
- register_failure_observer() (IPython.kernel.multiengineservice.QueuedEngine method), 486
- register_handler() (IPython.core.prefilter.PrefilterManager method), 388
- register_plugin() (IPython.core.plugin.PluginManager method), 360
- register_post_execute() (IPython.core.interactiveshell.InteractiveShell method), 315
- register_transformer() (IPython.core.prefilter.PrefilterManager method), 388
- reload() (in module IPython.lib.deeppreload), 590
- reload() (IPython.lib.demo.ClearDemo method), 595
- reload() (IPython.lib.demo.ClearIPDemo method), 596
- reload() (IPython.lib.demo.Demo method), 598
- reload() (IPython.lib.demo.IPythonDemo method), 600
- reload() (IPython.lib.demo.IPythonLineDemo method), 602
- reload() (IPython.lib.demo.LineDemo method), 603
- reload_extension() (IPython.core.extensions.ExtensionManager method), 250
- reload_history() (IPython.core.history.HistoryManager method), 272
- reload_history() (IPython.core.interactiveshell.InteractiveShell method), 315

remote_clear_pending_deferreds() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_keys() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_clear_properties() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_keys() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_clear_properties() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_kill() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_clear_queue() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_kill() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_del_properties() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_pull() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_del_properties() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_pull_function() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_execute() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_pull_function() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559
remote_execute() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559	remote_pull_function() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_get_client_name() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_pull_serialized() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_get_id() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_push() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_get_ids() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559	remote_push() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_get_pending_deferred() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_push_function() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_get_properties() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_push_function() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_get_properties() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_push_function() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_get_result() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_push_serialized() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
remote_get_result() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559	remote_push_serialized() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_has_properties() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462	remote_push_serialized() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559
remote_has_properties() (IPython.kernel.multienginefc.FCSynchronousMultiEngineFromMultiEngine method), 559	remote_register_engine() (IPython.kernel.enginefc.FCEngineReferenceFromService method), 462
	remote_register_engine() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 559

remote_reset() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 483

remote_set_id() (IPython.kernel.enginefc.FCEngineReferenceFromReference method), 462

remote_set_properties() (IPython.kernel.enginefc.FCEngineReferenceFromReference method), 463

remote_set_properties() (IPython.kernel.enginefc.FCEngineReferenceFromReference method), 463

remote_set_properties() (IPython.kernel.multienginefc.FCSynchronousMultiEngine method), 560

remove (IPython.kernel.core.util.InputList attribute), 458

remove (IPython.kernel.multiengineclient.QueueStatusList attribute), 555

remove (IPython.kernel.multiengineclient.ResultList attribute), 556

remove (IPython.utils.text.SList attribute), 680

remove() (IPython.lib.backgroundjobs.BackgroundJobManager method), 589

remove_all_observers() (IPython.utils.notification.NotificationCenter method), 663

remove_built_in() (IPython.core.builtin_trap.BuiltinTrap method), 219

remove_comments() (in module IPython.core.inputsplitters), 282

remove_pid_file() (IPython.kernel.ipclusterapp.IPClusterApp method), 499

remove_pid_file() (IPython.kernel.ipengineapp.IPEngineApp method), 504

rep_f() (in module IPython.core.history), 274

replace (IPython.utils.text.LSString attribute), 677

report() (in module IPython.testing.ipctest), 627

ReprDisplayFormatter (class in IPython.kernel.core.display_formatter), 433

reset() (IPython.core.debugger.Pdb method), 235

reset() (IPython.core.history.HistoryManager method), 272

reset() (IPython.core.inputsplitters.InputSplitter method), 281

reset() (IPython.core.inputsplitters.IPythonInputSplitter method), 279

reset() (IPython.core.interactiveshell.InteractiveShell method), 316

reset() (IPython.kernel.core.file_like.FileLike method), 437

reset() (IPython.kernel.core.interpreter.Interpreter method), 458

reset() (IPython.kernel.enginefc.EngineFromReference method), 462

reset() (IPython.kernel.engineservice.EngineService method), 470

reset() (IPython.kernel.engineservice.QueuedEngine method), 486

reset() (IPython.kernel.engineservice.ThreadedEngineService method), 486

reset() (IPython.kernel.multiengine.MultiEngine method), 541

reset() (IPython.kernel.multiengine.SynchronousMultiEngine method), 542

reset() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 548

reset() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngine method), 558

reset() (IPython.lib.demo.ClearDemo method), 595

reset() (IPython.lib.demo.ClearIPDemo method), 596

reset() (IPython.lib.demo.Demo method), 598

reset() (IPython.lib.demo.IPythonDemo method), 600

reset() (IPython.lib.demo.IPythonLineDemo method), 602

reset() (IPython.lib.demo.LineDemo method), 603

reset() (IPython.utils.autoattr.ResetMixin method), 644

resetbuffer() (IPython.core.interactiveshell.InteractiveShell method), 316

reset_selective() (IPython.core.interactiveshell.InteractiveShell method), 316

resetbuffer() (IPython.core.interactiveshell.InteractiveShell method), 316

ResetMixin (class in IPython.utils.autoattr), 644

restore_sys_module_state() (IPython.core.interactiveshell.InteractiveShell method), 316

result() (IPython.lib.backgroundjobs.BackgroundJobManager method), 589

ResultAlreadyRetrieved (class in IPython.kernel.error), 495

ResultList (class in IPython.kernel.multiengineclient), 555

ResultNotCompleted (class in IPython.kernel.error), 495

reverse (IPython.kernel.core.util.InputList attribute), 458

- ul style="list-style-type: none; padding-left: 0;">
- reverse (IPython.kernel.multiengineclient.QueueStatusList attribute), 555
- reverse (IPython.kernel.multiengineclient.ResultList attribute), 556
- reverse (IPython.utils.text.SList attribute), 680
- rfind (IPython.utils.text.LSString attribute), 677
- rindex (IPython.utils.text.LSString attribute), 677
- rjust (IPython.utils.text.LSString attribute), 677
- rlcomplete() (IPython.core.completer.IPCompleter method), 225
- RoundRobinMap (class in IPython.kernel.map), 506
- rpartition (IPython.utils.text.LSString attribute), 677
- rsplit (IPython.utils.text.LSString attribute), 677
- rstrip (IPython.utils.text.LSString attribute), 678
- run() (IPython.core.debugger.Pdb method), 235
- run() (IPython.core.history.HistorySaveThread method), 273
- run() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), 548
- run() (IPython.kernel.multiengineclient.FCFullSynchronousMultiEngineClient method), 558
- run() (IPython.kernel.twistedutil.ReactorInThread method), 583
- run() (IPython.lib.backgroundjobs.BackgroundJobBase method), 586
- run() (IPython.lib.backgroundjobs.BackgroundJobExpruncode() method), 586
- run() (IPython.lib.backgroundjobs.BackgroundJobFuncruncctx() method), 587
- Run() (IPython.lib.inputhookwx.EventLoopRunner method), 609
- run() (IPython.testing.ipctest.IPTester method), 627
- run_cell() (IPython.core.interactiveshell.InteractiveShell method), 316
- run_cell() (IPython.lib.demo.ClearDemo method), 595
- run_cell() (IPython.lib.demo.ClearIPDemo method), 596
- run_cell() (IPython.lib.demo.Demo method), 598
- run_cell() (IPython.lib.demo.IPythonDemo method), 600
- run_cell() (IPython.lib.demo.IPythonLineDemo method), 602
- run_cell() (IPython.lib.demo.LineDemo method), 603
- run_code() (IPython.core.interactiveshell.InteractiveShell method), 316
- run_file() (IPython.lib.irunner.InteractiveRunner method), 615
- run_file() (IPython.lib.irunner.IPythonRunner method), 614
- run_file() (IPython.lib.irunner.PythonRunner method), 616
- run_file() (IPython.lib.irunner.SAGERunner method), 617
- run_ipctest() (in module IPython.testing.ipctest), 627
- run_ipctestall() (in module IPython.testing.ipctest), 627
- run_one_block() (IPython.core.interactiveshell.InteractiveShell method), 317
- run_single_line() (IPython.core.interactiveshell.InteractiveShell method), 317
- run_source() (IPython.core.interactiveshell.InteractiveShell method), 317
- run_source() (IPython.lib.irunner.InteractiveRunner method), 615
- run_source() (IPython.lib.irunner.IPythonRunner method), 614
- run_source() (IPython.lib.irunner.PythonRunner method), 616
- run_source() (IPython.lib.irunner.SAGERunner method), 617
- runcall() (IPython.core.debugger.Pdb method), 235
- runctx() (IPython.core.interactiveshell.InteractiveShell method), 317
- runctx() (IPython.core.debugger.Pdb method), 235
- runCurrentCommand() (IPython.kernel.engineservice.QueuedEngine method), 486
- runeval() (IPython.core.debugger.Pdb method), 235
- runlines() (IPython.core.interactiveshell.InteractiveShell method), 318
- runner (IPython.testing.ipctest.IPTester attribute), 627
- RunnerFactory (class in IPython.lib.irunner), 617
- RunnerFactory (class in IPython.testing.mkdoctests), 631
- running (IPython.kernel.clientconnector.AsyncCluster attribute), 415
- running (IPython.kernel.clientconnector.Cluster attribute), 419
- runsourc() (IPython.core.interactiveshell.InteractiveShell method), 318
- s (IPython.utils.text.LSString attribute), 678
- s (IPython.utils.text.SList attribute), 680

[s_matches\(\)](#) (IPython.utils.strdispatch.StrDispatch method), [670](#)
[safe_execfile\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [318](#)
[safe_execfile_ipy\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [318](#)
[SafelyProcessEvent\(\)](#) (IPython.lib.inputhookwx.EventLoopTimer method), [611](#)
[SAGERunner](#) (class in IPython.lib.irunner), [617](#)
[save_history\(\)](#) (IPython.core.history.HistoryManager method), [272](#)
[save_history\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [319](#)
[save_pending_deferred\(\)](#) (IPython.kernel.multiengine.SynchronousMultiEngine method), [542](#)
[save_pending_deferred\(\)](#) (IPython.kernel.pendingdeferred.PendingDeferredManager method), [578](#)
[save_sys_module_state\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [319](#)
[saveResult\(\)](#) (IPython.kernel.engineservice.QueuedEngine method), [486](#)
[scatter\(\)](#) (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient method), [548](#)
[scatter\(\)](#) (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient method), [558](#)
[SecurityError](#) (class in IPython.kernel.error), [495](#)
[seek\(\)](#) (IPython.core.oinspect.myStringIO method), [351](#)
[seek\(\)](#) (IPython.lib.demo.ClearDemo method), [595](#)
[seek\(\)](#) (IPython.lib.demo.ClearIPDemo method), [596](#)
[seek\(\)](#) (IPython.lib.demo.Demo method), [598](#)
[seek\(\)](#) (IPython.lib.demo.IPythonDemo method), [600](#)
[seek\(\)](#) (IPython.lib.demo.IPythonLineDemo method), [602](#)
[seek\(\)](#) (IPython.lib.demo.LineDemo method), [604](#)
[separate_in](#) (IPython.core.interactiveshell.InteractiveShell attribute), [319](#)
[separate_out](#) (IPython.core.interactiveshell.InteractiveShell attribute), [319](#)
[separate_out2](#) (IPython.core.interactiveshell.InteractiveShell attribute), [319](#)
[SeparateStr](#) (class in IPython.core.interactiveshell), [322](#)
[SerializationError](#) (class in IPython.kernel.error), [496](#)
[serialize\(\)](#) (in module IPython.kernel.newserialized), [469](#)
[Serialized](#) (class in IPython.kernel.newserialized), [568](#)
[SerializeIt](#) (class in IPython.kernel.newserialized), [568](#)
[set\(\)](#) (IPython.core.display_trap.DisplayTrap method), [240](#)
[set\(\)](#) (IPython.kernel.core.display_trap.DisplayTrap method), [434](#)
[set\(\)](#) (IPython.kernel.core.output_trap.OutputTrap method), [448](#)
[set\(\)](#) (IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method), [453](#)
[set\(\)](#) (IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method), [454](#)
[set\(\)](#) (IPython.kernel.core.traceback_trap.TracebackTrap method), [455](#)
[set_active_scheme\(\)](#) (IPython.core.oinspect.Inspector method), [350](#)
[set_active_scheme\(\)](#) (IPython.core.oinspect.Inspector method), [350](#)
[set_color\(\)](#) (IPython.utils.coloransi.ColorSchemeTable method), [647](#)
[set_color\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), [319](#)
[set_break\(\)](#) (IPython.core.debugger.Pdb method), [235](#)
[set_colors\(\)](#) (IPython.core.debugger.Pdb method), [235](#)
[set_colors\(\)](#) (IPython.core.displayhook.DisplayHook method), [242](#)
[set_colors\(\)](#) (IPython.core.prompts.Prompt1 method), [397](#)
[set_colors\(\)](#) (IPython.core.prompts.Prompt2 method), [398](#)
[set_colors\(\)](#) (IPython.core.prompts.PromptOut method), [398](#)
[set_colors\(\)](#) (IPython.core.ultratb.AutoFormattedTB method), [402](#)
[set_colors\(\)](#) (IPython.core.ultratb.ColorTB method), [403](#)
[set_colors\(\)](#) (IPython.core.ultratb.FormattedTB method), [405](#)
[set_colors\(\)](#) (IPython.core.ultratb.ListTB method), [405](#)

406
set_colors() (IPython.core.ultratb.SyntaxTB method), 407
set_colors() (IPython.core.ultratb.TBTools method), 408
set_colors() (IPython.core.ultratb.VerboseTB method), 410
set_colors() (IPython.kernel.core.prompts.CachedOutprompt method), 450
set_colors() (IPython.kernel.core.prompts.Prompt1 method), 450
set_colors() (IPython.kernel.core.prompts.Prompt2 method), 451
set_colors() (IPython.kernel.core.prompts.PromptOut method), 451
set_command_line_config_log_level() (IPython.core.application.Application method), 215
set_command_line_config_log_level() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
set_command_line_config_log_level() (IPython.kernel.ipengineapp.IPEngineApp method), 504
set_completer_frame() (IPython.core.interactiveshell.InteractiveShell method), 319
set_continue() (IPython.core.debugger.Pdb method), 236
set_custom_completer() (IPython.core.interactiveshell.InteractiveShell method), 319
set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 319
set_default_config_log_level() (IPython.core.application.Application method), 215
set_default_config_log_level() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
set_default_config_log_level() (IPython.kernel.ipengineapp.IPEngineApp method), 504
set_default_value() (IPython.core.interactiveshell.SeparateStr method), 323
set_defaults() (IPython.config.loader.ArgumentParser method), 205
set_delims() (IPython.core.completer.CompletionSplitter method), 223
set_dynamic_initializer() (IPython.core.interactiveshell.SeparateStr method), 323
set_file_config_log_level() (IPython.core.application.Application method), 215
set_file_config_log_level() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
set_file_config_log_level() (IPython.kernel.ipengineapp.IPEngineApp method), 504
set_hook() (IPython.core.interactiveshell.InteractiveShell method), 320
set_id() (IPython.kernel.enginefc.EngineFromReference method), 462
set_inpthook() (IPython.lib.inpthook.InputHookManager method), 608
set_ip() (IPython.core.autocall.IPyAutocall method), 217
set_ip() (IPython.core.macro.Macro method), 326
set_metadata() (IPython.core.interactiveshell.SeparateStr method), 323
set_mode() (IPython.core.ultratb.AutoFormattedTB method), 402
set_mode() (IPython.core.ultratb.ColorTB method), 403
set_mode() (IPython.core.ultratb.FormattedTB method), 405
set_next() (IPython.core.debugger.Pdb method), 236
set_next_input() (IPython.core.interactiveshell.InteractiveShell method), 320
set_p_str() (IPython.core.prompts.BasePrompt method), 396
set_p_str() (IPython.core.prompts.Prompt1 method), 397
set_p_str() (IPython.core.prompts.Prompt2 method), 398
set_p_str() (IPython.core.prompts.PromptOut method), 398
set_p_str() (IPython.kernel.core.prompts.BasePrompt method), 449
set_p_str() (IPython.kernel.core.prompts.Prompt1 method), 450
set_p_str() (IPython.kernel.core.prompts.Prompt2 method), 451
set_p_str() (IPython.kernel.core.prompts.PromptOut method), 451

method), 451

set_properties() (IPython.kernel.enginefc.EngineFromRef default (IPython.utils.coloransi.ColorSchemeTable attribute), 647

method), 462

set_properties() (IPython.kernel.engineservice.EngineService default (IPython.utils.ipstruct.Struct attribute), 660

method), 470

set_properties() (IPython.kernel.engineservice.QueuedEngine default() (IPython.utils.pickleshare.PickleShareDB method), 667

method), 486

set_properties() (IPython.kernel.engineservice.ThreadedEngine default() (IPython.kernel.engineservice.Command method), 469

method), 488

set_properties() (IPython.kernel.multiengine.MultiEngine default() (IPython.lib.inputhookwx.EventLoopTimer method), 611

method), 541

set_properties() (IPython.kernel.multiengine.SynchronousMultiEngine default() (IPython.lib.inputhookwx.EventLoopTimer method), 611

method), 542

set_properties() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient setName() (IPython.core.history.HistorySaveThread method), 470

method), 548

set_properties() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient setName() (IPython.kernel.controllerservice.ControllerService method), 427

method), 558

set_quit() (IPython.core.debugger.Pdb method), 236

setName() (IPython.kernel.engineservice.EngineService method), 470

set_readline_completer() (IPython.core.interactiveshell.InteractiveShell setName() (IPython.kernel.engineservice.ThreadedEngineService method), 488

method), 320

setName() (IPython.kernel.twistedutil.ReactorInThread method), 583

set_return() (IPython.core.debugger.Pdb method), 236

setName() (IPython.lib.backgroundjobs.BackgroundJobBase method), 586

set_step() (IPython.core.debugger.Pdb method), 236

setName() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586

set_term_title() (in module IPython.utils.terminal), 673

setName() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587

set_trace() (IPython.core.debugger.Pdb method), 236

SetNextHandler() (IPython.lib.inputhookwx.EventLoopTimer method), 611

set_traps() (IPython.kernel.core.interpreter.Interpreter method), 443

SetOwner() (IPython.lib.inputhookwx.EventLoopTimer method), 611

set_until() (IPython.core.debugger.Pdb method), 236

SetPreviousHandler() (IPython.lib.inputhookwx.EventLoopTimer method), 611

setDaemon() (IPython.core.history.HistorySaveThread method), 273

setDaemon() (IPython.kernel.twistedutil.ReactorInThread method), 583

setDaemon() (IPython.lib.backgroundjobs.BackgroundJobBase method), 586

setDaemon() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586

setDaemon() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587

setdefault (IPython.config.loader.Config attribute), 206

setdefault (IPython.kernel.core.util.Bunch attribute), 457

setdefault (IPython.kernel.engineservice.StrictDict attribute), 487

setdefault (IPython.testing.globalipapp.ipnsdict attribute), 625

setdefault (IPython.kernel.controllerservice.ControllerServiceParent() (IPython.kernel.controllerservice.ControllerServiceParent() (IPython.kernel.engineservice.EngineService method), 470

setdefault (IPython.kernel.controllerservice.ThreadedEngineServiceParent() (IPython.kernel.engineservice.ThreadedEngineService method), 488

setdefault (IPython.kernel.clientinterfaces.IBlockingClientAd class method), 422

setdefault (IPython.kernel.clientinterfaces.IFCClientInterface class method), 425

setdefault (IPython.kernel.controllerservice.IControllerBase class method), 429

setdefault (IPython.kernel.controllerservice.IControllerCore class method), 429

class method), 432

setTaggedValue() (IPython.kernel.enginefc.IFCController class method), 465

setTaggedValue() (IPython.kernel.enginefc.IFCEngine class method), 467

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 472

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 475

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 477

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 480

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 482

setTaggedValue() (IPython.kernel.engineservice.IEngine class method), 484

setTaggedValue() (IPython.kernel.mapper.IMapper class method), 509

setTaggedValue() (IPython.kernel.mapper.IMultiEngineMapper class method), 511

setTaggedValue() (IPython.kernel.mapper.ITaskMapper class method), 513

setTaggedValue() (IPython.kernel.multiengine.IEngine class method), 518

setTaggedValue() (IPython.kernel.multiengine.IFullMultiEngine class method), 521

setTaggedValue() (IPython.kernel.multiengine.IFullSyncMultiEngine class method), 523

setTaggedValue() (IPython.kernel.multiengine.IMultiEngine class method), 525

setTaggedValue() (IPython.kernel.multiengine.IMultiEngine class method), 528

setTaggedValue() (IPython.kernel.multiengine.IMultiEngine class method), 530

setTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 532

setTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 535

setTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 537

setTaggedValue() (IPython.kernel.multiengine.ISynchronousMultiEngine class method), 540

setTaggedValue() (IPython.kernel.multiengineclient.IFCSystem class method), 551

setTaggedValue() (IPython.kernel.multiengineclient.IFCSystem class method), 553

setTaggedValue() (IPython.kernel.multienginefc.IFCSystem class method), 562

setTaggedValue() (IPython.kernel.newserialized.ISerialized class method), 565

setTaggedValue() (IPython.kernel.newserialized.IUnSerialized class method), 567

setTaggedValue() (IPython.kernel.parallelfunction.IMultiEngineParallel class method), 571

setTaggedValue() (IPython.kernel.parallelfunction.IParallelFunction class method), 574

setTaggedValue() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 576

setTaggedValue() (IPython.kernel.twistedutil.DeferredList class method), 582

setTaggedValue() (IPython.core.debugger.Pdb method), 236

setup_message() (IPython.kernel.core.interpreter.Interpreter class method), 443

setup_namespace() (IPython.kernel.core.history.InterpreterHistory method), 439

setup_namespace() (IPython.kernel.core.interpreter.Interpreter class method), 443

ShadowHist (class in IPython.core.history), 273

Shell (IPython.core.alias.AliasManager attribute), 211

Shell (IPython.core.builtin_trap.BuiltinTrap attribute), 219

Shell (IPython.core.displayhook.DisplayHook attribute), 242

Shell (IPython.core.extensions.ExtensionManager attribute), 250

Shell (IPython.core.prefilter.AliasChecker attribute), 362

Shell (IPython.core.prefilter.AliasHandler attribute), 364

Shell (IPython.core.prefilter.AssignMagicTransformer attribute), 365

Shell (IPython.core.prefilter.AssignmentChecker attribute), 368

Shell (IPython.core.prefilter.AssignSystemTransformer attribute), 366

Shell (IPython.core.prefilter.AutocallChecker attribute), 371

Shell (IPython.core.prefilter.AutoHandler attribute), 369

Shell (IPython.core.prefilter.AutoMagicChecker attribute), 370

Shell (IPython.core.prefilter.EmacsChecker attribute), 373

Shell (IPython.core.prefilter.EmacsHandler attribute), 373

- tribute), 374
- shell (IPython.core.prefilter.EscCharsChecker attribute), 375
- shell (IPython.core.prefilter.HelpHandler attribute), 377
- shell (IPython.core.prefilter.IPyAutocallChecker attribute), 378
- shell (IPython.core.prefilter.IPyPromptTransformer attribute), 379
- shell (IPython.core.prefilter.MagicHandler attribute), 381
- shell (IPython.core.prefilter.MultiLineMagicChecker attribute), 382
- shell (IPython.core.prefilter.PrefilterChecker attribute), 384
- shell (IPython.core.prefilter.PrefilterHandler attribute), 385
- shell (IPython.core.prefilter.PrefilterManager attribute), 388
- shell (IPython.core.prefilter.PrefilterTransformer attribute), 390
- shell (IPython.core.prefilter.PyPromptTransformer attribute), 391
- shell (IPython.core.prefilter.PythonOpsChecker attribute), 392
- shell (IPython.core.prefilter.ShellEscapeChecker attribute), 394
- shell (IPython.core.prefilter.ShellEscapeHandler attribute), 395
- ShellEscapeChecker (class in IPython.core.prefilter), 393
- ShellEscapeHandler (class in IPython.core.prefilter), 394
- shlex_split() (in module IPython.core.completerlib), 227
- show() (IPython.lib.demos.ClearDemo method), 595
- show() (IPython.lib.demos.ClearIPDemo method), 596
- show() (IPython.lib.demos.Demo method), 598
- show() (IPython.lib.demos.IPythonDemo method), 600
- show() (IPython.lib.demos.IPythonLineDemo method), 602
- show() (IPython.lib.demos.LineDemo method), 604
- show_all() (IPython.lib.demos.ClearDemo method), 595
- show_all() (IPython.lib.demos.ClearIPDemo method), 596
- show_all() (IPython.lib.demos.Demo method), 598
- show_all() (IPython.lib.demos.IPythonDemo method), 600
- show_all() (IPython.lib.demos.IPythonLineDemo method), 602
- show_all() (IPython.lib.demos.LineDemo method), 604
- show_exception_only() (IPython.core.ultratb.AutoFormattedTB method), 402
- show_exception_only() (IPython.core.ultratb.ColorTB method), 403
- show_exception_only() (IPython.core.ultratb.FormattedTB method), 405
- show_exception_only() (IPython.core.ultratb.ListTB method), 406
- show_exception_only() (IPython.core.ultratb.SyntaxTB method), 407
- show_hidden() (in module IPython.utils.wildcard), 685
- show_in_pager() (in module IPython.core.hooks), 277
- show_usage() (IPython.core.interactiveshell.InteractiveShell method), 320
- showdiff() (in module IPython.utils.upgradedir), 683
- showsyntaxerror() (IPython.core.interactiveshell.InteractiveShell method), 320
- showtraceback() (IPython.core.interactiveshell.InteractiveShell method), 320
- shutdown_hook() (in module IPython.core.hooks), 277
- sigint_handler() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
- SimpleMessageCache (class in IPython.kernel.core.message_cache), 447
- single_dir_expand() (in module IPython.core.completer), 226
- singleton_printers (IPython.core.formatters.BaseFormatter attribute), 253
- singleton_printers (IPython.core.formatters.HTMLFormatter attribute), 258
- singleton_printers (IPython.core.formatters.JSONFormatter attribute), 260
- singleton_printers (IPython.core.formatters.LatexFormatter

- attribute), 262
- singleton_printers (IPython.core.formatters.PlainTextFormatter method), 444
- attribute), 267
- singleton_printers (IPython.core.formatters.PNGFormatter method), 223
- attribute), 264
- singleton_printers (IPython.core.formatters.SVGFormatter method), 283
- attribute), 270
- skip() (in module IPython.testing.decorators), 622
- skip() (in module IPython.testing.decorators_trial), 623
- skipif() (in module IPython.testing.decorators), 622
- skipif() (in module IPython.testing.decorators_trial), 624
- sleep_deferred() (in module IPython.kernel.twistedutil), 583
- SList (class in IPython.utils.text), 679
- snip_print() (in module IPython.core.page), 354
- soft_define_alias() (IPython.core.alias.AliasManager method), 211
- softspace() (in module IPython.core.interactiveshell), 324
- sort (IPython.kernel.core.util.InputList attribute), 458
- sort (IPython.kernel.multiengineclient.QueueStatusList attribute), 555
- sort (IPython.kernel.multiengineclient.ResultList attribute), 556
- sort() (IPython.utils.text.SList method), 680
- sort_checkers() (IPython.core.prefilter.PrefilterManager method), 388
- sort_compare() (in module IPython.utils.data), 648
- sort_transformers() (IPython.core.prefilter.PrefilterManager method), 388
- source_raw_reset() (IPython.core.inputsplitter.IPythonInputSplitter method), 279
- source_reset() (IPython.core.inputsplitter.InputSplitter method), 281
- source_reset() (IPython.core.inputsplitter.IPythonInputSplitter method), 279
- SpaceInInput (class in IPython.core.interactiveshell), 323
- split (IPython.utils.text.LSString attribute), 678
- split_blocks() (in module IPython.core.inputsplitter), 283
- split_blocks() (IPython.core.inputsplitter.InputSplitter method), 281
- split_blocks() (IPython.core.inputsplitter.IPythonInputSplitter method), 279
- split_commands() (IPython.kernel.core.interpreter.Interpreter method), 444
- split_line() (IPython.core.completer.CompletionSplitter method), 223
- split_user_input() (in module IPython.core.inputsplitter), 283
- split_user_input() (in module IPython.core.splitinput), 399
- splitlines (IPython.utils.text.LSString attribute), 678
- spstr (IPython.utils.text.LSString attribute), 678
- spstr (IPython.utils.text.SList attribute), 681
- start() (in module IPython.utils.growl), 653
- start() (IPython.core.application.Application method), 215
- start() (IPython.core.history.HistorySaveThread method), 273
- start() (IPython.kernel.clientconnector.AsyncCluster method), 415
- start() (IPython.kernel.clientconnector.Cluster method), 419
- start() (IPython.kernel.core.fd_redirector.FDRedirector method), 436
- start() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
- start() (IPython.kernel.ipengineapp.IPEngineApp method), 504
- start() (IPython.kernel.twistedutil.ReactorInThread method), 583
- start() (IPython.lib.backgroundjobs.BackgroundJobBase method), 586
- start() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 586
- start() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 587
- Start() (IPython.lib.inputhookwx.EventLoopTimer method), 611
- start_app() (IPython.core.application.Application method), 215
- start_app() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
- start_app() (IPython.kernel.ipengineapp.IPEngineApp method), 504
- start_app_start() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
- start_app_stop() (IPython.kernel.ipclusterapp.IPClusterApp method), 500
- start_controller() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

start_displayhook() (IPython.core.displayhook.DisplayHook method), 242

start_engines() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

start_event_loop_qt4() (in module IPython.lib.guisupport), 605

start_event_loop_wx() (in module IPython.lib.guisupport), 605

start_ipython() (in module IPython.testing.globalipapp), 626

start_launchers() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

start_logging() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

start_logging() (IPython.kernel.ipengineapp.IEngineApp method), 504

start_mpi() (IPython.kernel.ipengineapp.IEngineApp method), 504

startService() (IPython.kernel.controllerservice.ControllerService method), 427

startService() (IPython.kernel.engineservice.EngineService method), 470

startService() (IPython.kernel.engineservice.ThreadedEngineService method), 488

startswith (IPython.utils.text.LSString attribute), 678

startup_message() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

status() (IPython.lib.backgroundjobs.BackgroundJobManager method), 589

stb2text() (IPython.core.ultratb.AutoFormattedTB method), 402

stb2text() (IPython.core.ultratb.ColorTB method), 403

stb2text() (IPython.core.ultratb.FormattedTB method), 405

stb2text() (IPython.core.ultratb.ListTB method), 406

stb2text() (IPython.core.ultratb.SyntaxTB method), 407

stb2text() (IPython.core.ultratb.TBTools method), 409

stb2text() (IPython.core.ultratb.VerboseTB method), 410

stdin_ready() (in module IPython.lib.inputhookwx), 612

stop() (IPython.core.history.HistorySaveThread method), 273

stop() (IPython.kernel.clientconnector.AsyncClusterHook method), 415

stop() (IPython.kernel.clientconnector.ClusterHook method), 419

stop() (IPython.kernel.core.fd_redirector.FDRedirector method), 436

stop() (IPython.kernel.twistedutil.ReactorInThread method), 583

Stop() (IPython.lib.inputhookwx.EventLoopTimer method), 611

stop_controller() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

stop_engines() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

stop_here() (IPython.core.debugger.Pdb method), 236

stop_launchers() (IPython.kernel.ipclusterapp.IPClusterApp method), 500

StopLocalExecution (class in IPython.kernel.error), 496

stopService() (IPython.kernel.controllerservice.ControllerService method), 427

stopService() (IPython.kernel.engineservice.EngineService method), 470

stopService() (IPython.kernel.engineservice.ThreadedEngineService method), 488

store_inputs() (IPython.core.history.HistoryManager method), 272

str_safe() (in module IPython.core.prompts), 399

str_safe() (in module IPython.kernel.core.prompts), 452

StrDispatch (class in IPython.utils.strdispatch), 670

stress() (in module IPython.utils.pickleshare), 668

StrictDict (class in IPython.kernel.engineservice), 486

strip (IPython.utils.text.LSString attribute), 678

Struct (class in IPython.utils.ipstruct), 657

structured_traceback() (IPython.core.ultratb.AutoFormattedTB method), 402

structured_traceback() (IPython.core.ultratb.ColorTB method), 403

structured_traceback() (IPython.core.ultratb.FormattedTB method), 405

structured_traceback() (IPython.core.ultratb.ListTB method), 406

structured_traceback() (IPython.core.ultratb.SyntaxTB method), 407

408

structured_traceback()
(IPython.core.ultratb.TBTools method),
409

structured_traceback()
(IPython.core.ultratb.VerboseTB method),
410

subDict() (IPython.kernel.engineservice.StrictDict
method), 487

submitCommand() (IPython.kernel.engineservice.QueueSubmitter
method), 486

subscribe() (IPython.kernel.clientinterfaces.IBlockingClientAdapter
class method), 422

subscribe() (IPython.kernel.clientinterfaces.IFCCClientInterface
class method), 425

subscribe() (IPython.kernel.controllerservice.IControllerBase
class method), 429

subscribe() (IPython.kernel.controllerservice.IControllerBase
class method), 432

subscribe() (IPython.kernel.enginefc.IFCCControllerBase
class method), 465

subscribe() (IPython.kernel.enginefc.IFCEngine
class method), 468

subscribe() (IPython.kernel.engineservice.IEngineBases
class method), 473

subscribe() (IPython.kernel.engineservice.IEngineCores
class method), 475

subscribe() (IPython.kernel.engineservice.IEngineProperties
class method), 477

subscribe() (IPython.kernel.engineservice.IEngineQueue
class method), 480

subscribe() (IPython.kernel.engineservice.IEngineSerializers
class method), 482

subscribe() (IPython.kernel.engineservice.IEngineThreaded
class method), 484

subscribe() (IPython.kernel.mapper.IMapper class
method), 509

subscribe() (IPython.kernel.mapper.IMultiEngineMapperFactory
class method), 511

subscribe() (IPython.kernel.mapper.ITaskMapperFactory
class method), 513

subscribe() (IPython.kernel.multiengine.IEngineMultiplexer
class method), 518

subscribe() (IPython.kernel.multiengine.IFullMultiEngine
class method), 521

subscribe() (IPython.kernel.multiengine.IFullSynchronousMultiEngine
class method), 523

subscribe() (IPython.kernel.multiengine.IMultiEngine
class method), 525

subscribe() (IPython.kernel.multiengine.IMultiEngineCoordinator
class method), 528

subscribe() (IPython.kernel.multiengine.IMultiEngineExtras
class method), 530

subscribe() (IPython.kernel.multiengine.ISynchronousEngineMultiplexer
class method), 532

subscribe() (IPython.kernel.multiengine.ISynchronousMultiEngine
class method), 535

subscribe() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator
class method), 537

subscribe() (IPython.kernel.multiengine.ISynchronousMultiEngineExtras
class method), 540

subscribe() (IPython.kernel.multiengineclient.IFullBlockingMultiEngine
class method), 551

subscribe() (IPython.kernel.multiengineclient.IPendingResult
class method), 553

subscribe() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine
class method), 562

subscribe() (IPython.kernel.newserialized.ISerialized
class method), 565

subscribe() (IPython.kernel.newserialized.IUnSerialized
class method), 567

subscribe() (IPython.kernel.parallelfunction.IMultiEngineParallelDecorator
class method), 571

subscribe() (IPython.kernel.parallelfunction.IParallelFunction
class method), 574

subscribe() (IPython.kernel.parallelfunction.ITaskParallelDecorator
class method), 576

SVGFormatter (class in IPython.core.formatters),
268

switch_log() (IPython.utils.text.LSString attribute), 678

switch_log() (IPython.core.logger.Logger method),
325

sync_inputs() (IPython.core.history.HistoryManager
method), 272

synchronize_with_editor() (in module
IPython.core.hooks), 277

SynchronousMultiEngine (class in
IPython.kernel.multiengine), 541

SynchronousTaskMapper (class in
IPython.kernel.mapper), 514

syncProperties() (IPython.kernel.enginefc.EngineFromReference
method), 462

SyncTracebackTrap (class in
IPython.kernel.core.sync_traceback_trap),
453

SyntaxTB (class in IPython.core.ultratb), 407

sys_info() (in module IPython.utils.sysinfo), [672](#)
 system() (IPython.core.interactiveshell.InteractiveShell
 method), [321](#)
 system_shell() (in module IPython.kernel.core.util),
[459](#)

T

target_outdated() (in module IPython.utils.path), [665](#)
 target_update() (in module IPython.utils.path), [665](#)
 tarModule() (in module IPython.kernel.util), [584](#)
 TaskAborted (class in IPython.kernel.error), [496](#)
 TaskMapper (class in IPython.kernel.mapper), [515](#)
 TaskRejectError (class in IPython.kernel.error), [496](#)
 TaskTimeout (class in IPython.kernel.error), [497](#)
 TBTools (class in IPython.core.ultratb), [408](#)
 tearDown() (IPython.testing.tools.TempFileMixin
 method), [638](#)
 Tee (class in IPython.utils.io), [655](#)
 tell() (IPython.core.oinspect.myStringIO method),
[352](#)
 temp_pyfile() (in module IPython.utils.io), [656](#)
 TempFileMixin (class in IPython.testing.tools), [638](#)
 term_clear() (in module IPython.utils.terminal), [673](#)
 TermColors (class in IPython.utils.coloransi), [648](#)
 test() (in module IPython.testing), [621](#)
 test() (in module IPython.utils.pickleshare), [668](#)
 test_for() (in module IPython.testing.ipctest), [628](#)
 test_trivial() (in module
 IPython.testing.plugin.test_refs), [637](#)
 text() (IPython.core.ultratb.AutoFormattedTB
 method), [402](#)
 text() (IPython.core.ultratb.ColorTB method), [403](#)
 text() (IPython.core.ultratb.FormattedTB method),
[405](#)
 text() (IPython.core.ultratb.ListTB method), [407](#)
 text() (IPython.core.ultratb.SyntaxTB method), [408](#)
 text() (IPython.core.ultratb.TBTools method), [409](#)
 text() (IPython.core.ultratb.VerboseTB method), [410](#)
 thisown (IPython.lib.inputhookwx.EventLoopTimer
 attribute), [612](#)
 ThreadedEngineService (class in
 IPython.kernel.engineservice), [487](#)
 timing() (in module IPython.utils.timing), [682](#)
 timings() (in module IPython.utils.timing), [683](#)
 timings_out() (in module IPython.utils.timing), [683](#)
 title (IPython.utils.text.LSString attribute), [678](#)
 tkinter_clipboard_get() (in module
 IPython.lib.clipboard), [589](#)
 to_work_dir() (IPython.kernel.ipclusterapp.IPClusterApp
 method), [500](#)
 to_work_dir() (IPython.kernel.ipengineapp.IPEngineApp
 method), [504](#)
 toggle_set_term_title() (in module
 IPython.utils.terminal), [673](#)
 trace_dispatch() (IPython.core.debugger.Pdb
 method), [236](#)
 traceback() (IPython.lib.backgroundjobs.BackgroundJobBase
 method), [586](#)
 traceback() (IPython.lib.backgroundjobs.BackgroundJobExpr
 method), [586](#)
 traceback() (IPython.lib.backgroundjobs.BackgroundJobFunc
 method), [587](#)
 traceback() (IPython.lib.backgroundjobs.BackgroundJobManager
 method), [589](#)
 TracebackTrap (class in
 IPython.kernel.core.traceback_trap),
[455](#)
 Tracer (class in IPython.core.debugger), [236](#)
 trait_metadata() (IPython.core.alias.AliasManager
 method), [211](#)
 trait_metadata() (IPython.core.builtin_trap.BuiltinTrap
 method), [219](#)
 trait_metadata() (IPython.core.display_trap.DisplayTrap
 method), [240](#)
 trait_metadata() (IPython.core.displayhook.DisplayHook
 method), [242](#)
 trait_metadata() (IPython.core.displaypub.DisplayPublisher
 method), [245](#)
 trait_metadata() (IPython.core.extensions.ExtensionManager
 method), [250](#)
 trait_metadata() (IPython.core.formatters.BaseFormatter
 method), [253](#)
 trait_metadata() (IPython.core.formatters.DisplayFormatter
 method), [255](#)
 trait_metadata() (IPython.core.formatters.HTMLFormatter
 method), [258](#)
 trait_metadata() (IPython.core.formatters.JSONFormatter
 method), [260](#)
 trait_metadata() (IPython.core.formatters.LatexFormatter
 method), [262](#)
 trait_metadata() (IPython.core.formatters.PlainTextFormatter
 method), [267](#)
 trait_metadata() (IPython.core.formatters.PNGFormatter
 method), [265](#)
 trait_metadata() (IPython.core.formatters.SVGFormatter
 method), [270](#)

trait_metadata() (IPython.core.interactiveshell.InteractiveShell method), 321	trait_metadata() (IPython.core.prefilter.PyPromptTransformer method), 391
trait_metadata() (IPython.core.payload.PayloadManager method), 356	trait_metadata() (IPython.core.prefilter.PythonOpsChecker method), 392
trait_metadata() (IPython.core.plugin.Plugin method), 359	trait_metadata() (IPython.core.prefilter.ShellEscapeChecker method), 394
trait_metadata() (IPython.core.plugin.PluginManager method), 360	trait_metadata() (IPython.core.prefilter.ShellEscapeHandler method), 395
trait_metadata() (IPython.core.prefilter.AliasChecker method), 362	trait_names() (IPython.core.alias.AliasManager method), 211
trait_metadata() (IPython.core.prefilter.AliasHandler method), 364	trait_names() (IPython.core.builtin_trap.BuiltinTrap method), 219
trait_metadata() (IPython.core.prefilter.AssignMagicTransformer method), 365	transform() (IPython.core.display_trap.DisplayTrap method), 240
trait_metadata() (IPython.core.prefilter.AssignmentChecker method), 368	transform_names() (IPython.core.displayhook.DisplayHook method), 242
trait_metadata() (IPython.core.prefilter.AssignSystemTransformer method), 366	transform_names() (IPython.core.displaypub.DisplayPublisher method), 245
trait_metadata() (IPython.core.prefilter.AutocallChecker method), 372	transform_names() (IPython.core.extensions.ExtensionManager method), 250
trait_metadata() (IPython.core.prefilter.AutoHandler method), 369	transform_names() (IPython.core.formatters.BaseFormatter method), 253
trait_metadata() (IPython.core.prefilter.AutoMagicChecker method), 370	transform_names() (IPython.core.formatters.DisplayFormatter method), 255
trait_metadata() (IPython.core.prefilter.EmacsChecker method), 373	transform_names() (IPython.core.formatters.HTMLFormatter method), 258
trait_metadata() (IPython.core.prefilter.EmacsHandler method), 374	transform_names() (IPython.core.formatters.JSONFormatter method), 260
trait_metadata() (IPython.core.prefilter.EscCharsChecker method), 375	transform_names() (IPython.core.formatters.LatexFormatter method), 262
trait_metadata() (IPython.core.prefilter.HelpHandler method), 377	transform_names() (IPython.core.formatters.PlainTextFormatter method), 267
trait_metadata() (IPython.core.prefilter.IPyAutocallChecker method), 378	transform_names() (IPython.core.formatters.PNGFormatter method), 265
trait_metadata() (IPython.core.prefilter.IPyPromptTransformer method), 379	transform_names() (IPython.core.formatters.SVGFormatter method), 270
trait_metadata() (IPython.core.prefilter.MagicHandler method), 381	transform_names() (IPython.core.interactiveshell.InteractiveShell method), 321
trait_metadata() (IPython.core.prefilter.MultiLineMagicChecker method), 383	transform_names() (IPython.core.payload.PayloadManager method), 356
trait_metadata() (IPython.core.prefilter.PrefilterChecker method), 384	transform_names() (IPython.core.plugin.Plugin method), 359
trait_metadata() (IPython.core.prefilter.PrefilterHandler method), 385	transform_names() (IPython.core.plugin.PluginManager method), 360
trait_metadata() (IPython.core.prefilter.PrefilterManager method), 388	transform_names() (IPython.core.prefilter.AliasChecker method), 363
trait_metadata() (IPython.core.prefilter.PrefilterTransformer method), 390	transform_names() (IPython.core.prefilter.AliasHandler method), 364

[trait_names\(\) \(IPython.core.prefilter.AssignMagicTransformer method\), 365](#)
[trait_names\(\) \(IPython.core.prefilter.AssignmentChecker method\), 368](#)
[trait_names\(\) \(IPython.core.prefilter.AssignSystemTransformer method\), 366](#)
[trait_names\(\) \(IPython.core.prefilter.AutocallChecker method\), 372](#)
[trait_names\(\) \(IPython.core.prefilter.AutoHandler method\), 369](#)
[trait_names\(\) \(IPython.core.prefilter.AutoMagicChecker method\), 370](#)
[trait_names\(\) \(IPython.core.prefilter.EmacsChecker method\), 373](#)
[trait_names\(\) \(IPython.core.prefilter.EmacsHandler method\), 374](#)
[trait_names\(\) \(IPython.core.prefilter.EscCharsChecker method\), 376](#)
[trait_names\(\) \(IPython.core.prefilter.HelpHandler method\), 377](#)
[trait_names\(\) \(IPython.core.prefilter.IPyAutocallChecker method\), 378](#)
[trait_names\(\) \(IPython.core.prefilter.IPyPromptTransformer method\), 379](#)
[trait_names\(\) \(IPython.core.prefilter.MagicHandler method\), 381](#)
[trait_names\(\) \(IPython.core.prefilter.MultiLineMagicChecker method\), 383](#)
[trait_names\(\) \(IPython.core.prefilter.PrefilterChecker method\), 384](#)
[trait_names\(\) \(IPython.core.prefilter.PrefilterHandler method\), 385](#)
[trait_names\(\) \(IPython.core.prefilter.PrefilterManager method\), 388](#)
[trait_names\(\) \(IPython.core.prefilter.PrefilterTransformer method\), 390](#)
[trait_names\(\) \(IPython.core.prefilter.PyPromptTransformer method\), 391](#)
[trait_names\(\) \(IPython.core.prefilter.PythonOpsChecker method\), 393](#)
[trait_names\(\) \(IPython.core.prefilter.ShellEscapeChecker method\), 394](#)
[trait_names\(\) \(IPython.core.prefilter.ShellEscapeHandler method\), 395](#)
[traits\(\) \(IPython.core.alias.AliasManager method\), 211](#)
[traits\(\) \(IPython.core.builtin_trap.BuiltinTrap method\), 219](#)
[traits\(\) \(IPython.core.display_trap.DisplayTrap method\), 240](#)
[traits\(\) \(IPython.core.displayhook.DisplayHook method\), 242](#)
[traits\(\) \(IPython.core.displaypub.DisplayPublisher method\), 245](#)
[traits\(\) \(IPython.core.extensions.ExtensionManager method\), 250](#)
[traits\(\) \(IPython.core.formatters.BaseFormatter method\), 253](#)
[traits\(\) \(IPython.core.formatters.DisplayFormatter method\), 255](#)
[traits\(\) \(IPython.core.formatters.HTMLFormatter method\), 258](#)
[traits\(\) \(IPython.core.formatters.JSONFormatter method\), 260](#)
[traits\(\) \(IPython.core.formatters.LatexFormatter method\), 262](#)
[traits\(\) \(IPython.core.formatters.PlainTextFormatter method\), 267](#)
[traits\(\) \(IPython.core.formatters.PNGFormatter method\), 265](#)
[traits\(\) \(IPython.core.formatters.SVGFormatter method\), 270](#)
[traits\(\) \(IPython.core.interactiveshell.InteractiveShell method\), 321](#)
[traits\(\) \(IPython.core.payload.PayloadManager method\), 356](#)
[traits\(\) \(IPython.core.plugin.Plugin method\), 359](#)
[traits\(\) \(IPython.core.plugin.PluginManager method\), 360](#)
[traits\(\) \(IPython.core.prefilter.AliasChecker method\), 363](#)
[traits\(\) \(IPython.core.prefilter.AliasHandler method\), 364](#)
[traits\(\) \(IPython.core.prefilter.AssignMagicTransformer method\), 365](#)
[traits\(\) \(IPython.core.prefilter.AssignmentChecker method\), 368](#)
[traits\(\) \(IPython.core.prefilter.AssignSystemTransformer method\), 366](#)
[traits\(\) \(IPython.core.prefilter.AutocallChecker method\), 372](#)
[traits\(\) \(IPython.core.prefilter.AutoHandler method\), 369](#)
[traits\(\) \(IPython.core.prefilter.AutoMagicChecker method\), 370](#)
[traits\(\) \(IPython.core.prefilter.EmacsChecker](#)

method), 373

traits() (IPython.core.prefilter.EmacsHandler method), 374

traits() (IPython.core.prefilter.EscCharsChecker method), 376

traits() (IPython.core.prefilter.HelpHandler method), 377

traits() (IPython.core.prefilter.IPyAutocallChecker method), 378

traits() (IPython.core.prefilter.IPyPromptTransformer method), 379

traits() (IPython.core.prefilter.MagicHandler method), 381

traits() (IPython.core.prefilter.MultiLineMagicChecker method), 383

traits() (IPython.core.prefilter.PrefilterChecker method), 384

traits() (IPython.core.prefilter.PrefilterHandler method), 386

traits() (IPython.core.prefilter.PrefilterManager method), 388

traits() (IPython.core.prefilter.PrefilterTransformer method), 390

traits() (IPython.core.prefilter.PyPromptTransformer method), 391

traits() (IPython.core.prefilter.PythonOpsChecker method), 393

traits() (IPython.core.prefilter.ShellEscapeChecker method), 394

traits() (IPython.core.prefilter.ShellEscapeHandler method), 395

transform() (IPython.core.prefilter.AssignMagicTransformer method), 365

transform() (IPython.core.prefilter.AssignSystemTransformer method), 367

transform() (IPython.core.prefilter.IPyPromptTransformer method), 380

transform() (IPython.core.prefilter.PrefilterTransformer method), 390

transform() (IPython.core.prefilter.PyPromptTransformer method), 391

transform_alias() (IPython.core.alias.AliasManager method), 211

transform_assign_magic() (in module IPython.core.inputsplitter), 284

transform_assign_system() (in module IPython.core.inputsplitter), 284

transform_classic_prompt() (in module IPython.core.inputsplitter), 284

transform_ipy_prompt() (in module IPython.core.inputsplitter), 284

transform_line() (IPython.core.prefilter.PrefilterManager method), 389

transformers (IPython.core.prefilter.PrefilterManager attribute), 389

translate (IPython.utils.text.LSString attribute), 678

truncate() (IPython.core.oinspect.myStringIO method), 352

truncate() (IPython.kernel.core.file_like.FileLike method), 437

try_import() (in module IPython.core.completerlib), 227

TryNext (class in IPython.core.error), 247

two_phase() (in module IPython.kernel.pendingdeferred), 578

type_printers (IPython.core.formatters.BaseFormatter attribute), 253

type_printers (IPython.core.formatters.HTMLFormatter attribute), 258

type_printers (IPython.core.formatters.JSONFormatter attribute), 260

type_printers (IPython.core.formatters.LatexFormatter attribute), 263

type_printers (IPython.core.formatters.PlainTextFormatter attribute), 268

type_printers (IPython.core.formatters.PNGFormatter attribute), 265

type_printers (IPython.core.formatters.SVGFormatter attribute), 270

unbin() (IPython.lib.inpthookwx.EventLoopTimer method), 611

uncache() (IPython.utils.pickleshare.PickleShareDB method), 667

uncan() (in module IPython.kernel.pickleutil), 580

uncanDict() (in module IPython.kernel.pickleutil), 580

uncanSequence() (in module IPython.kernel.pickleutil), 580

undefine_alias() (IPython.core.alias.AliasManager method), 211

uniq_stable() (in module IPython.utils.data), 648

Unlink() (IPython.lib.inpthookwx.EventLoopTimer method), 611

[unload_extension\(\) \(IPython.core.extensions.ExtensionManager method\), 455](#)
[method\), 250](#)
[unpackage\(\) \(IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient method\), 558](#)
[unpackageFailure\(\) \(in module IPython.kernel.pbutil\), 577](#)
[unpause\(\) \(IPython.kernel.twistedutil.DeferredList method\), 582](#)
[UnpickableException \(class in IPython.kernel.error\), 497](#)
[unquote_ends\(\) \(in module IPython.utils.text\), 682](#)
[unregister_checker\(\) \(IPython.core.prefilter.PrefilterManager method\), 389](#)
[unregister_engine\(\) \(IPython.kernel.controllerservice.ControllerAdapterBase method\), 427](#)
[unregister_engine\(\) \(IPython.kernel.controllerservice.ControllerSession method\), 427](#)
[unregister_engine\(\) \(IPython.kernel.multiengine.MultiEngine class method\), 541](#)
[unregister_failure_observer\(\) \(IPython.kernel.engineservice.QueuedEngineUnsubscribe method\), 486](#)
[unregister_handler\(\) \(IPython.core.prefilter.PrefilterManager method\), 389](#)
[unregister_plugin\(\) \(IPython.core.plugin.PluginManager method\), 360](#)
[unregister_transformer\(\) \(IPython.core.prefilter.PrefilterManager method\), 389](#)
[unserialize\(\) \(in module IPython.kernel.newserialized\), 569](#)
[UnSerialized \(class in IPython.kernel.newserialized\), 568](#)
[UnSerializeIt \(class in IPython.kernel.newserialized\), 568](#)
[unset\(\) \(IPython.core.display_trap.DisplayTrap method\), 240](#)
[unset\(\) \(IPython.kernel.core.display_trap.DisplayTrap method\), 434](#)
[unset\(\) \(IPython.kernel.core.output_trap.OutputTrap method\), 448](#)
[unset\(\) \(IPython.kernel.core.redirector_output_trap.RedirectorOutputTrap method\), 453](#)
[unset\(\) \(IPython.kernel.core.sync_traceback_trap.SyncTracebackTrap method\), 454](#)
[unset\(\) \(IPython.kernel.core.traceback_trap.TracebackTrap method\), 454](#)
[unset_traps\(\) \(IPython.kernel.core.interpreter.Interpreter method\), 454](#)
[unsubscribe\(\) \(IPython.kernel.clientinterfaces.IBlockingClientAdapter class method\), 422](#)
[unsubscribe\(\) \(IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method\), 425](#)
[unsubscribe\(\) \(IPython.kernel.controllerservice.IControllerBase class method\), 429](#)
[unsubscribe\(\) \(IPython.kernel.controllerservice.IControllerCore class method\), 432](#)
[unsubscribe\(\) \(IPython.kernel.enginefc.IFCControllerBase class method\), 465](#)
[unsubscribe\(\) \(IPython.kernel.enginefc.IFCEngine class method\), 468](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineBase class method\), 473](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineCore class method\), 475](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineProperties class method\), 477](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineQueued class method\), 480](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineSerialized class method\), 482](#)
[unsubscribe\(\) \(IPython.kernel.engineservice.IEngineThreaded class method\), 484](#)
[unsubscribe\(\) \(IPython.kernel.mapper.IMapper class method\), 509](#)
[unsubscribe\(\) \(IPython.kernel.mapper.IMultiEngineMapperFactory class method\), 511](#)
[unsubscribe\(\) \(IPython.kernel.mapper.ITaskMapperFactory class method\), 513](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IEngineMultiplexer class method\), 518](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IFullMultiEngine class method\), 521](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine class method\), 523](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IMultiEngine class method\), 525](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IMultiEngineCoordinator class method\), 528](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.IMultiEngineExtras class method\), 530](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.ISynchronousEngineMulti class method\), 532](#)
[unsubscribe\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 532](#)

class method), 535

unsubscribe() (IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator class method), 537

unsubscribe() (IPython.kernel.multiengine.ISynchronousMultiEnginesCoordinator class method), 540

unsubscribe() (IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient class method), 551

unsubscribe() (IPython.kernel.multiengineclient.IPendingResult class method), 553

unsubscribe() (IPython.kernel.multienginefc.IFCSynchronousMultiEngine class method), 562

unsubscribe() (IPython.kernel.newserialized.ISerialized class method), 565

unsubscribe() (IPython.kernel.newserialized.IUnSerialized class method), 567

unsubscribe() (IPython.kernel.parallelfunction.IMultiEngineParallelCoordinator class method), 571

unsubscribe() (IPython.kernel.parallelfunction.IParallelValidator class method), 574

unsubscribe() (IPython.kernel.parallelfunction.ITaskParallelDecorator class method), 576

update (IPython.config.loader.Config attribute), 206

update (IPython.kernel.core.util.Bunch attribute), 457

update (IPython.utils.coloransi.ColorSchemeTable attribute), 647

update (IPython.utils.ipstruct.Struct attribute), 660

update() (IPython.kernel.core.prompts.CachedOutput method), 450

update() (IPython.kernel.engineservice.StrictDict method), 487

update() (IPython.testing.globalipapp.ipnsdict method), 625

update() (IPython.utils.pickleshare.PickleShareDB method), 667

update_history() (IPython.kernel.core.history.InterpreterHistory method), 439

update_user_ns() (IPython.core.displayhook.DisplayHook method), 242

upgrade_dir() (in module IPython.utils.upgradedir), 683

upper (IPython.utils.text.LSString attribute), 679

usage (IPython.core.application.Application attribute), 216

UsageError (class in IPython.core.error), 247

user_aliases (IPython.core.alias.AliasManager attribute), 211

user_call() (IPython.core.debugger.Pdb method), 236

user_multiengine() (IPython.core.debugger.Pdb method), 236

user_multiengines() (IPython.core.interactiveshell.InteractiveShell method), 321

user_multiengine() (IPython.core.debugger.Pdb method), 236

user_multiengine() (IPython.core.interactiveshell.InteractiveShell method), 321

validate() (IPython.core.interactiveshell.SeparateStr method), 323

validate() (IPython.core.alias.AliasManager method), 211

validateInvariants() (IPython.kernel.clientinterfaces.IBlockingClientA class method), 422

validateInvariants() (IPython.kernel.clientinterfaces.IFCCClientInterface class method), 425

validateInvariants() (IPython.kernel.controllerservice.IControllerBase class method), 429

validateInvariants() (IPython.kernel.controllerservice.IControllerCore class method), 432

validateInvariants() (IPython.kernel.enginefc.IFCCControllerBase class method), 465

validateInvariants() (IPython.kernel.enginefc.IFCEngine class method), 468

validateInvariants() (IPython.kernel.engineservice.IEngineBase class method), 473

validateInvariants() (IPython.kernel.engineservice.IEngineCore class method), 475

validateInvariants() (IPython.kernel.engineservice.IEngineProperties class method), 477

validateInvariants() (IPython.kernel.engineservice.IEngineQueued class method), 480

validateInvariants() (IPython.kernel.engineservice.IEngineSerialized class method), 482

validateInvariants() (IPython.kernel.engineservice.IEngineThreaded class method), 484

validateInvariants() (IPython.kernel.mapper.IMapper class method), 509

validateInvariants() (IPython.kernel.mapper.IMultiEngineMapperFac class method), 511

validateInvariants() (IPython.kernel.mapper.ITaskMapperFactory class method), 513

[validateInvariants\(\) \(IPython.kernel.multiengine.IEngineMultiEngine class method\), 518](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.IFullMultiEngine class method\), 521](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine class method\), 523](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.IMultiEngine class method\), 525](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.IMultiEngineColor class method\), 528](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.IMultiEngineExtras class method\), 530](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 532](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine class method\), 535](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineColor class method\), 537](#)
[validateInvariants\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineExtras class method\), 540](#)
[validateInvariants\(\) \(IPython.kernel.multiengineclient.IFullBlockingClient class method\), 551](#)
[validateInvariants\(\) \(IPython.kernel.multiengineclient.IPendingResponse class method\), 553](#)
[validateInvariants\(\) \(IPython.kernel.multienginefc.IFCSynchro class method\), 562](#)
[validateInvariants\(\) \(IPython.kernel.newserialized.ISerialized class method\), 565](#)
[validateInvariants\(\) \(IPython.kernel.newserialized.IUnSerialized class method\), 567](#)
[validateInvariants\(\) \(IPython.kernel.parallelfunction.IMultiEngine class method\), 571](#)
[validateInvariants\(\) \(IPython.kernel.parallelfunction.IParallelFunction class method\), 574](#)
[validateInvariants\(\) \(IPython.kernel.parallelfunction.ITaskQueue class method\), 576](#)
[values \(IPython.config.loader.Config attribute\), 207](#)
[values \(IPython.kernel.core.util.Bunch attribute\), 457](#)
[values \(IPython.kernel.engineservice.StrictDict attribute\), 487](#)
[values \(IPython.testing.globalipapp.ipnsdict attribute\), 626](#)
[values \(IPython.utils.coloransi.ColorSchemeTable attribute\), 647](#)
[values \(IPython.utils.ipstruct.Struct attribute\), 660](#)
[values\(\) \(IPython.utils.pickleshare.PickleShareDB method\), 667](#)
[wait_for_file\(\) \(in module IPython.kernel.core.util\), 583](#)
[waitget\(\) \(IPython.utils.pickleshare.PickleShareDB method\), 667](#)
[warn\(\) \(in module IPython.utils.warn\), 684](#)
[weakref\(\) \(IPython.kernel.clientinterfaces.IBlockingClientAdaptor class method\), 422](#)
[weakref\(\) \(IPython.kernel.clientinterfaces.IFCClientInterfaceProvider class method\), 425](#)
[weakref\(\) \(IPython.kernel.controllerservice.IControllerBase class method\), 430](#)
[weakref\(\) \(IPython.kernel.controllerservice.IControllerCore class method\), 432](#)
[weakref\(\) \(IPython.kernel.enginefc.IFCControllerBase class method\), 465](#)
[weakref\(\) \(IPython.kernel.enginefc.IFCEngine class method\), 468](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineBase class method\), 473](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineCore class method\), 475](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineProperties class method\), 477](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineQueued class method\), 480](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineSerialized class method\), 482](#)
[weakref\(\) \(IPython.kernel.engineservice.IEngineThreaded class method\), 485](#)
[weakref\(\) \(IPython.kernel.mapper.IMapper class method\), 509](#)
[weakref\(\) \(IPython.kernel.mapper.IMultiEngineMapperFactory class method\), 511](#)

[weakref\(\) \(IPython.kernel.mapper.ITaskMapperFactory.wrapResultList\(\) \(in module class method\), 513](#)
[weakref\(\) \(IPython.kernel.multiengine.IEngineMultiplexer.write\(\) \(IPython.core.interactiveshell.InteractiveShell class method\), 519](#)
[weakref\(\) \(IPython.kernel.multiengine.IFullMultiEngine.write\(\) \(IPython.core.oinspect.myStringIO method\), class method\), 521](#)
[weakref\(\) \(IPython.kernel.multiengine.IFullSynchronousMultiEngine.write\(\) \(IPython.core.prompts.BasePrompt method\), class method\), 523](#)
[weakref\(\) \(IPython.kernel.multiengine.IMultiEngine.write\(\) \(IPython.core.prompts.Prompt1 method\), class method\), 526](#)
[weakref\(\) \(IPython.kernel.multiengine.IMultiEngineCoordinator.write\(\) \(IPython.core.prompts.Prompt2 method\), class method\), 528](#)
[weakref\(\) \(IPython.kernel.multiengine.IMultiEngineExecutor.write\(\) \(IPython.core.prompts.PromptOut method\), class method\), 530](#)
[weakref\(\) \(IPython.kernel.multiengine.ISynchronousEngine.writeMultiPrompt\(\) \(IPython.kernel.core.prompts.BasePrompt class method\), 532](#)
[weakref\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngine.write\(\) \(IPython.kernel.core.prompts.Prompt1 class method\), 535](#)
[weakref\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineCoordinator.write\(\) \(IPython.kernel.core.prompts.Prompt2 class method\), 537](#)
[weakref\(\) \(IPython.kernel.multiengine.ISynchronousMultiEngineExecutor.write\(\) \(IPython.kernel.core.prompts.PromptOut class method\), 540](#)
[weakref\(\) \(IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient.write\(\) \(IPython.testing.mkdoctests.IndentOut class method\), 551](#)
[weakref\(\) \(IPython.kernel.multiengineclient.IPendingResult.write\(\) \(IPython.utils.io.IOStream method\), 654](#)
[weakref\(\) \(IPython.kernel.multiengineclient.IFullBlockingMultiEngineClient.write\(\) \(IPython.utils.io.Tee method\), 655](#)
[weakref\(\) \(IPython.kernel.multiengineclient.IFullSynchronousMultiEngineClient.write\(\) \(IPython.core.interactiveshell.InteractiveShell class method\), 562](#)
[weakref\(\) \(IPython.kernel.newserialized.ISerialized.write_format_data\(\) \(IPython.core.displayhook.DisplayHook class method\), 565](#)
[weakref\(\) \(IPython.kernel.newserialized.IUnSerialized.write_output_prompt\(\) \(IPython.core.displayhook.DisplayHook class method\), 567](#)
[weakref\(\) \(IPython.kernel.parallelfunction.IMultiEngineParallelFunction.write\(\) \(IPython.core.displayhook.DisplayHook class method\), 571](#)
[weakref\(\) \(IPython.kernel.parallelfunction.IParallelFunction.write_payload\(\) \(IPython.core.payload.PayloadManager class method\), 574](#)
[weakref\(\) \(IPython.kernel.parallelfunction.ITaskParallelFunction.write_pid_file\(\) \(IPython.kernel.ipclusterapp.IPClusterApp class method\), 576](#)
[wildcards_case_sensitive \(IPython.core.interactiveshell.InteractiveShell attribute\), 322](#)
[win32_clipboard_get\(\) \(in module IPython.lib.clipboard\), 589](#)
[with_obj\(\) \(in module IPython.utils.attic\), 642](#)
[wrap_deprecated\(\) \(in module IPython.utils.attic\), 643](#)
[wrapped_execute\(\) \(IPython.kernel.engineservice.ThreadedEngineService method\), 489](#)

X

xmode (IPython.core.interactiveshell.InteractiveShell
attribute), [322](#)

xsys() (in module IPython.testing.globalipapp), [626](#)

Z

zfill (IPython.utils.text.LSString attribute), [679](#)

zip_pull() (IPython.kernel.multiengineclient.FullBlockingMultiEngineClient
method), [548](#)

zip_pull() (IPython.kernel.multienginefc.FCFullSynchronousMultiEngineClient
method), [558](#)