

---

# **IPython Documentation**

*Release 0.11.dev*

**The IPython Development Team**

April 07, 2011



# CONTENTS



# 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.ipplib` 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 Foolscape 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 Foolscape 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 Foolscape 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 2.6 or 2.7. There is an experimental port of IPython for Python3 [on GitHub](#)

**Warning:** Officially, IPython supports Python versions 2.6 and 2.7.  
IPython 0.11 has a hard syntax dependency on 2.6, and will no longer work on Python  $\leq 2.5$ .

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[ zmq, test]
```

This will get `pyzmq`, which is 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 setup.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 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`.

### 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.parallel (parallel computing)

`IPython.kernel` has been replaced by `IPython.parallel`, which uses ZeroMQ for all communication.

`IPython.parallel` provides a nice architecture for parallel computing. The main focus of this architecture is on interactive parallel computing. These features require just one package: `pyzmq`. See the next section for `pyzmq` details.

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

```
$ easy_install ipython[zmq]      # will include pyzmq
```

Security in `IPython.parallel` is provided by SSH tunnels. By default, Linux and OSX clients will use the shell `ssh` command, but on Windows, we also support tunneling with `paramiko` [[paramiko](#)].

## 3.6 Dependencies for IPython.zmq

### 3.6.1 pyzmq

IPython 0.11 introduced some new functionality, including a two-process execution model using ZeroMQ for communication [[ZeroMQ](#)]. The Python bindings to ZeroMQ are found in the `pyzmq` project, which is `easy_install`-able once you have ZeroMQ installed (or even if you don't).

`IPython.zmq` depends on `pyzmq` `>= 2.0.10.1`, but `IPython.parallel` requires the more recent 2.1.4. 2.1.4 also has binary releases for OSX and Windows, that do not require prior installation of `libzmq`.

## 3.7 Dependencies for ipython-qtconsole (new GUI)

### 3.7.1 PyQt

Also with 0.11, a new GUI was added using the work in `IPython.zmq`, which can be launched with `ipython-qtconsole`. The GUI is built on PyQt, which can be installed from the [PyQt website](#).

### 3.7.2 pygments

The syntax-highlighting in `ipython-qtconsole` is done with the `pygments` project, which is `easy_install`-able.



---

# 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 is typically installed in the `IPYTHON_DIR` directory. For Linux users, this will be `$HOME/.config/ipython`, and for other users it will be `$HOME/.ipython`. For Windows users, `$HOME` resolves to `C:\Documents and Settings\YourUserName` in most instances.

### 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. \$IPYTHON\_DIR/ipythonrc : load basic things you always want.

2. `$IPYTHON_DIR/ipythonrc-math` : load (1) and basic math-related modules.
3. `$IPYTHON_DIR/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 `dreload` 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 beings 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_DIR/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

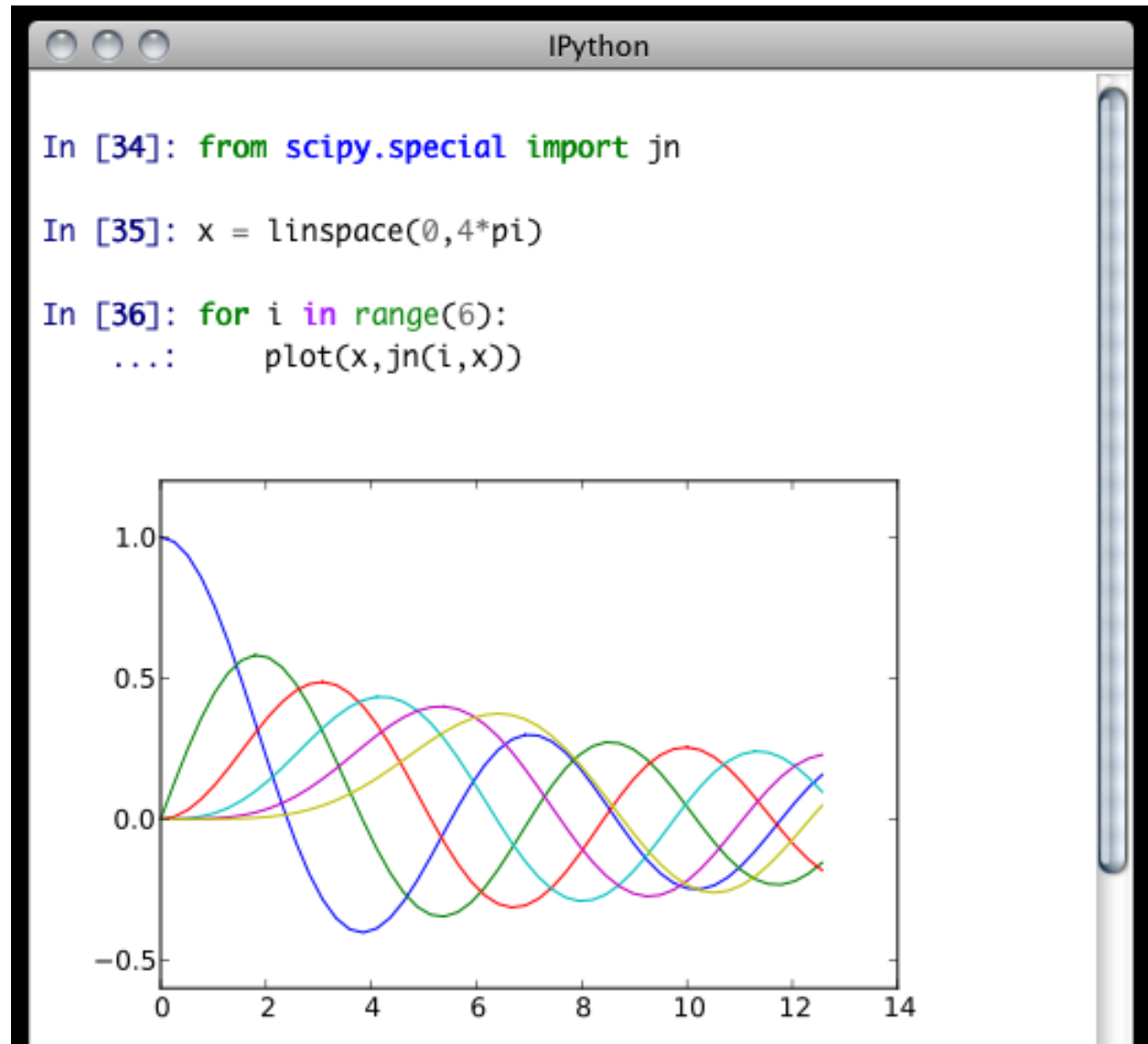
The %loadpy magic has been added, just for the GUI frontend. It takes any python script (must end in ‘.py’), and pastes its contents as your next input, so you can edit it before executing. The script may be on your machine, but you can also specify a url, and it will download the script from the web.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

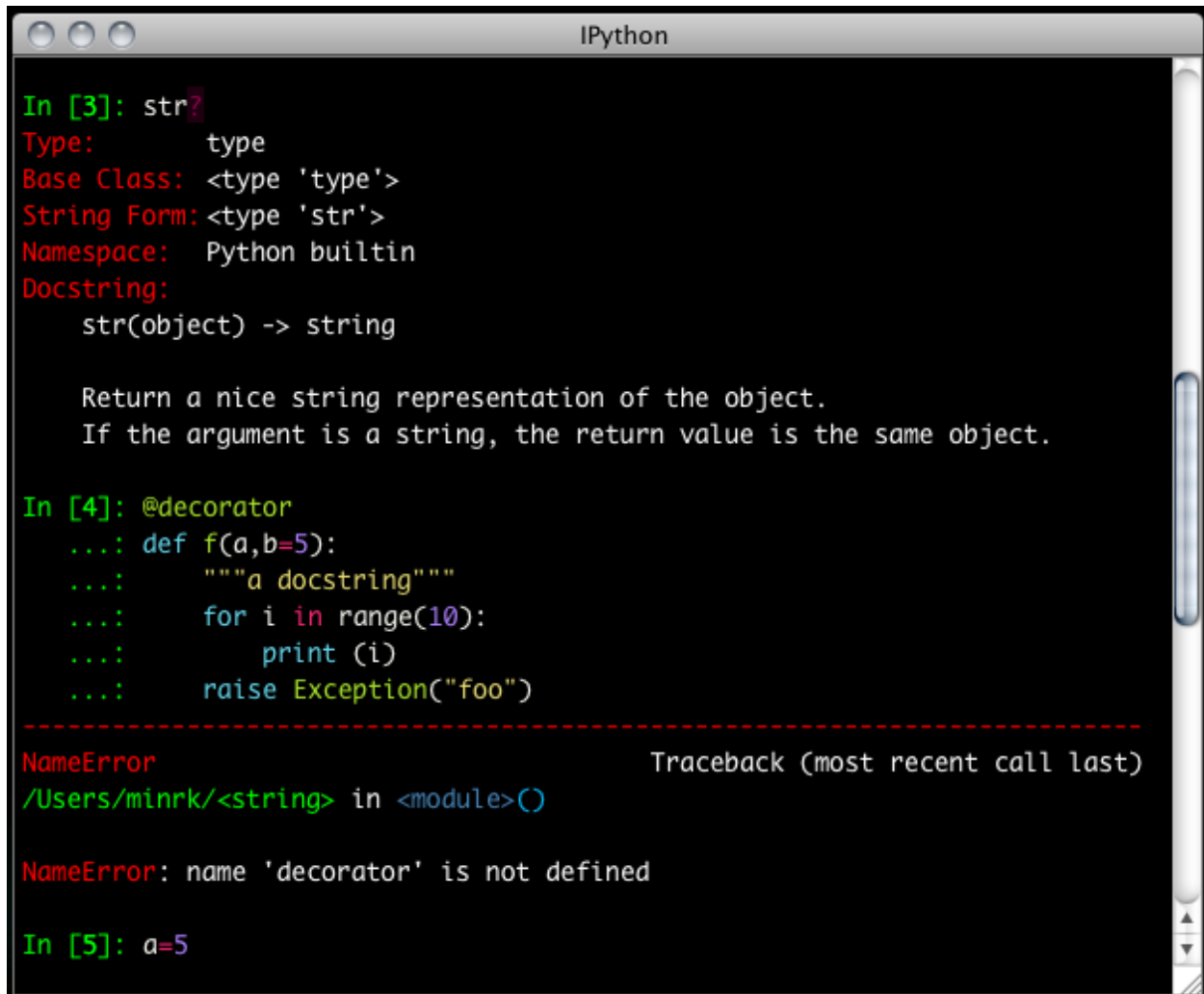
Terminal IPython has always had some coloring, but never syntax highlighting. There are a few simple color choices, specified by the `--colors` flag or `%colors` magic:

- `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 four components:

- The IPython engine.
- The IPython hub.
- The IPython schedulers.
- The controller client.

These components live in the `IPython.parallel` 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 processes provide an interface for working with a set of engines. At a general level, the controller is a collection of processes to which IPython engines and clients can connect. The controller is composed of a `Hub` and a collection of `Schedulers`. These `Schedulers` are typically run in separate processes but on the same machine as the `Hub`, but can be run anywhere from local threads or on remote machines.

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, all of these models are implemented via the client's `View.apply()` method, with various arguments, or constructing `View` objects to represent subsets of engines. The two primary models for interacting with engines are:

- A **Direct** interface, where engines are addressed explicitly.
- A **LoadBalanced** interface, where the Scheduler is trusted with assigning work to appropriate engines.

Advanced users can readily extend the `View` models to enable other styles of parallelism.

**Note:** A single controller and set of engines can be used with multiple models simultaneously. This opens the door for lots of interesting things.

---

## The Hub

The center of an IPython cluster is the Hub. This is the process that keeps track of engine connections, schedulers, clients, as well as all task requests and results. The primary role of the Hub is to facilitate queries of the cluster state, and minimize the necessary information required to establish the many connections involved in connecting new clients and engines.

## Schedulers

All actions that can be performed on the engine go through a Scheduler. While the engines themselves block when user code is run, the schedulers hide that from the user to provide a fully asynchronous interface to a set of engines.

## IPython client and views

There is one primary object, the `Client`, for connecting to a cluster. For each execution model, there is a corresponding `View`. These views allow users to interact with a set of engines through the interface. Here are the two default views:

- The `DirectView` class for explicit addressing.
- The `LoadBalancedView` class for destination-agnostic scheduling.

## Security

IPython uses ZeroMQ for networking, which has provided many advantages, but one of the setbacks is its utter lack of security [ZeroMQ]. By default, no IPython connections are encrypted, but open ports only listen on localhost. The only source of security for IPython is via ssh-tunnel. IPython supports both shell (*openssh*) and *paramiko* based tunnels for connections. There is a key necessary to submit requests, but due to the lack of encryption, it does not provide significant security if loopback traffic is compromised.

In our architecture, the controller is the only process that listens on network ports, and is thus the main point of vulnerability. The standard model for secure connections is to designate that the controller listen on localhost, and use ssh-tunnels to connect clients and/or engines.

To connect and authenticate to the controller an engine or client needs some information that the controller has stored in a JSON file. Thus, the JSON files need to be copied to a location where the clients and engines can find them. Typically, this is the `~/.ipython/cluster_default/security` directory on the host where the client/engine is running (which could be a different host than the controller). Once the JSON files are copied over, everything should work fine.

Currently, there are two JSON files that the controller creates:

**ipcontroller-engine.json** This JSON file has the information necessary for an engine to connect to a controller.

**ipcontroller-client.json** The client's connection information. This may not differ from the engine's, but since the controller may listen on different ports for clients and engines, it is stored separately.

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

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

**Warning:** Even at its most secure, the Controller listens on ports on localhost, and every time you make a tunnel, you open a localhost port on the connecting machine that points to the Controller. If localhost on the Controller's machine, or the machine of any client or engine, is untrusted, then your Controller is insecure. There is no way around this with ZeroMQ.

### 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 start -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.parallel import Client

In [2]: c = Client()

In [4]: c.ids
Out[4]: set([0, 1, 2, 3])

In [5]: c[:].apply_sync(lambda : "Hello, World")
Out[5]: [ 'Hello, World', 'Hello, World', 'Hello, World', 'Hello, World' ]
```

When a client is created with no arguments, the client tries to find the corresponding JSON file in the local `~/ipython/cluster_default/security` directory. Or if you specified a profile, you can use that with the Client. This should cover most cases:

```
In [2]: c = Client(profile='myprofile')
```

If you have put the JSON file in a different location or it has a different name, create the client like this:

```
In [2]: c = Client('/path/to/my/ipcontroller-client.json')
```

Remember, a client needs to be able to see the Hub's ports to connect. So if they are on a different machine, you may need to use an ssh server to tunnel access to that machine, then you would connect to it with:

```
In [2]: c = Client(sshtserver='myhub.example.com')
```

Where ‘myhub.example.com’ is the url or IP address of the machine on which the Hub process is running (or another machine that has direct access to the Hub’s ports).

The SSH server may already be specified in `ipcontroller-client.json`, if the controller was instructed at its launch time.

You are now ready to learn more about the *Direct* and *LoadBalanced* interfaces 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 JSON file (`ipcontroller-engine.json`) 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 JSON file (`ipcontroller-engine.json`) is located.

At this point, the controller and engines will be connected. By default, the JSON files created by the controller are put into the `~/.ipython/cluster_default/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 to actually use the running controller from a client is to move the JSON file `ipcontroller-client.json` from `host0` to any host where clients will be run. If these file are put into the `~/.ipython/cluster_default/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 (or other *qsub* systems, such as SGE).
4. When the controller is started on localhost and the engines are started on remote nodes using **ssh**.
5. When engines are started using the Windows HPC Server batch system.

---

**Note:** Currently **ipcluster** requires that the `~/ .ipython/cluster_<profile>/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.

---

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

The simplest way to use **ipcluster** requires no configuration, and will launch a controller and a number of engines on the local machine. For instance, to start one controller and 4 engines on localhost, just do:

```
$ ipcluster start -n 4
```

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

```
$ ipcluster -h
```

### 5.2.3 Configuring an IPython cluster

Cluster configurations are stored as *profiles*. You can create a new profile with:

```
$ ipcluster create -p myprofile
```

This will create the directory `IPYTHONDIR/cluster_myprofile`, and populate it with the default configuration files for the three IPython cluster commands. Once you edit those files, you can continue to call **ipcluster/ipcontroller/ipengine** with no arguments beyond `-p myprofile`, and any configuration will be maintained.

There is no limit to the number of profiles you can have, so you can maintain a profile for each of your common use cases. The default profile will be used whenever the profile argument is not specified, so edit `IPYTHONDIR/cluster_default/*_config.py` to represent your most common use case.

The configuration files are loaded with commented-out settings and explanations, which should cover most of the available possibilities.

## Using various batch systems with ipcluster

**ipcluster** has a notion of Launchers that can start controllers and engines with various remote execution schemes. Currently supported models include *mpiexec*, PBS-style (Torque, SGE), and Windows HPC Server.

---

**Note:** The Launchers and configuration are designed in such a way that advanced users can subclass and configure them to fit their own system that we have not yet supported (such as Condor)

---

## 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.

If these are satisfied, you can create a new profile:

```
$ ipcluster create -p mpi
```

and edit the file `IPYTHONDIR/cluster_mpi/ipcluster_config.py`.

There, instruct *ipcluster* to use the *MPIExec* launchers by adding the lines:

```
c.Global.engine_launcher = 'IPython.parallel.apps.launcher.MPIExecEngineSetLauncher'
```

If the default MPI configuration is correct, then you can now start your cluster, with:

```
$ ipcluster start -n 4 -p mpi
```

This does the following:

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

If you have a reason to also start the Controller with *mpi*, you can specify:

```
c.Global.controller_launcher = 'IPython.parallel.apps.launcher.MPIExecControllerLauncher'
```

---

**Note:** The Controller *will not* be in the same MPI universe as the engines, so there is not much reason to do this unless sysadmins demand it.

---

On newer MPI implementations (such as OpenMPI), this will work even if you don't make any calls to `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 specify the `c.MPI.use` option in `ipengine_config.py`:

```
c.MPI.use = '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.

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.

As usual, we will start by creating a fresh profile:

```
$ ipcluster create -p pbs
```

And in `ipcluster_config.py`, we will select the PBS launchers for the controller and engines:

```
c.Global.controller_launcher = 'IPython.parallel.apps.launcher.PBSControllerLauncher'
c.Global.engine_launcher = 'IPython.parallel.apps.launcher.PBSEngineSetLauncher'
```

IPython does provide simple default batch templates for PBS and SGE, but you may need to specify your own. 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 $queue

cd $$PBS_O_WORKDIR
export PATH=$$HOME/usr/local/bin
export PYTHONPATH=$$HOME/usr/local/lib/python2.7/site-packages
/usr/local/bin/mpirun -n ${n} ipengine --cluster_dir=${cluster_dir}
```

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. There will always be a `${n}` and `${cluster_dir}` variable passed to the template. These allow the batch system to know how many engines, and where the configuration files reside. The same is true for the batch queue, with the template variable `$queue`.
3. Because `$` 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, or in SGE, where the config lines start with `#$`, which will have to be `##$`.
4. Any options to **ipengine** can be given in the batch script template, or in `ipengine_config.py`.
5. Depending on the configuration of your system, you may have to set environment variables in the script template.

The controller template should be similar, but simpler:



```
#PBS -N ipython
#PBS -j oe
#PBS -l walltime=00:10:00
#PBS -l nodes=1:ppn=4
#PBS -q $queue

cd $$PBS_O_WORKDIR
export PATH=$$HOME/usr/local/bin
export PYTHONPATH=$$HOME/usr/local/lib/python2.7/site-packages
ipcontroller --cluster_dir=${cluster_dir}
```

Once you have created these scripts, save them with names like `pbs.engine.template`. Now you can load them into the `ipcluster_config` with:

```
c.PBSEngineSetLauncher.batch_template_file = "pbs.engine.template"

c.PBSControllerLauncher.batch_template_file = "pbs.controller.template"
```

Alternately, you can just define the templates as strings inside `ipcluster_config`.

Whether you are using your own templates or our defaults, the extra configurables available are the number of engines to launch (`$n`, and the batch system queue to which the jobs are to be submitted (`$queue`)). These are configurables, and can be specified in `ipcluster_config`:

```
c.PBSLauncher.queue = 'veryshort.q'
c.PBSEngineSetLauncher.n = 64
```

Note that assuming you are running PBS on a multi-node cluster, the Controller's default behavior of listening only on localhost is likely too restrictive. In this case, also assuming the nodes are safely behind a firewall, you can simply instruct the Controller to listen for connections on all its interfaces, by adding in `ipcontroller_config`:

```
c.RegistrationFactory.ip = '*'
```

You can now run the cluster with:

```
$ ipcluster start -p pbs -n 128
```

Additional configuration options can be found in the PBS section of `ipcluster_config`.

---

**Note:** Due to the flexibility of configuration, the PBS launchers work with simple changes to the template for other **qsub**-using systems, such as Sun Grid Engine, and with further configuration in similar batch systems like Condor.

---

## Using ipcluster in SSH mode

The SSH mode uses **ssh** to execute **ipengine** on remote nodes and **ipcontroller** can be run remotely as well, or on localhost.

---

**Note:** 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.

---

As usual, we start by creating a clean profile:

```
$ ipcluster create -p ssh
```

To use this mode, select the SSH launchers in `ipcluster_config.py`:

```
c.Global.engine_launcher = 'IPython.parallel.apps.launcher.SSHEngineSetLauncher'
# and if the Controller is also to be remote:
c.Global.controller_launcher = 'IPython.parallel.apps.launcher.SSHControllerLauncher'
```

The controller's remote location and configuration can be specified:

```
# Set the user and hostname for the controller
# c.SSHControllerLauncher.hostname = 'controller.example.com'
# c.SSHControllerLauncher.user = os.environ.get('USER', 'username')

# Set the arguments to be passed to ipcontroller
# note that remotely launched ipcontroller will not get the contents of
# the local ipcontroller_config.py unless it resides on the *remote host*
# in the location specified by the --cluster_dir argument.
# c.SSHControllerLauncher.program_args = ['-r', '-ip', '0.0.0.0', '--cluster_dir', '/path/']
```

---

**Note:** SSH mode does not do any file movement, so you will need to distribute configuration files manually. To aid in this, the `reuse_files` flag defaults to `True` for ssh-launched Controllers, so you will only need to do this once, unless you override this flag back to `False`.

---

Engines are specified in a dictionary, by hostname and the number of engines to be run on that host.

```
c.SSHEngineSetLauncher.engines = { 'host1.example.com' : 2,
                                   'host2.example.com' : 5,
                                   'host3.example.com' : (1, ['--cluster_dir', '/home/different/location']),
                                   'host4.example.com' : 8 }
```

- The *engines* dict, where the keys are the host we want to run engines on and the value is the number of engines to run on that host.
- on *host3*, the value is a tuple, where the number of engines is first, and the arguments to be passed to **ipengine** are the second element.

For engines without explicitly specified arguments, the default arguments are set in a single location:

```
c.SSHEngineSetLauncher.engine_args = ['--cluster_dir', '/path/to/cluster_ssh']
```

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.
- No file movement -

## 5.2.4 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 JSON files in `~/.ipython/cluster_default/security`. You are now ready to use the controller and engines from IPython.

**Warning:** The order of the above operations may be important. You *must* start the controller before the engines, unless you are reusing connection information (via `-r`), in which case ordering is not important.

---

**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.json` from `~/.ipython/cluster_<profile>/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.json` file is located. There are two ways you can do this:

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

The `--file` flag works like this:

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

---

**Note:** If the controller's and engine's hosts all have a shared file system (~/.ipython/cluster\_<profile>/security is the same on all of them), then things will just work!

---

### Make JSON files persistent

At first glance it may seem that managing the JSON files is a bit annoying. Going back to the house and key analogy, copying the JSON 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 JSON file) once, and then simply use it at any point in the future.

To do this, the only thing you have to do is specify the `-r` flag, so that the connection information in the JSON files remains accurate:

```
$ ipcontroller -r
```

Then, just copy the JSON 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 reuse the file.

---

**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/cluster\_<profile>/log. Sending the log files to us will often help us to debug any problems.

### Configuring *ipcontroller*

#### Ports and addresses

#### Database Backend

See Also:

### Configuring *ipengine*

---

**Note:** TODO

---

## 5.3 IPython's Direct interface

The direct, or 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 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 start -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 Client instance

The first step is to import the IPython `IPython.parallel` module and then create a `Client` instance:

```
In [1]: from IPython.parallel import Client
```

```
In [2]: rc = Client()
```

This form assumes that the default connection information (stored in `ipcontroller-client.json` found in `IPYTHON_DIR/cluster_default/security`) is accurate. If the controller was started on a remote machine, you must copy that connection file to the client machine, or enter its contents as arguments to the `Client` constructor:

```
# If you have copied the json connector file from the controller:
In [2]: rc = Client('/path/to/ipcontroller-client.json')
# or to connect with a specific profile you have set up:
In [3]: rc = Client(profile='mpi')
```

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

```
In [3]: rc.ids
Out[3]: [0, 1, 2, 3]
```

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

For direct execution, we will make use of a `DirectView` object, which can be constructed via list-access to the client:

```
In [4]: dview = rc[:] # use all engines
```

#### See Also:

For more information, see the in-depth explanation of *Views*.

### 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 client interface provides a simple way of accomplishing this: using the `DirectView`'s `map()` method.

#### 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, since IPython's interface is all about functions anyway, you can just use the builtin `map()` with a `RemoteFunction`, or a `DirectView`'s `map()` method:

```
In [62]: serial_result = map(lambda x:x*10, range(32))
```

```
In [63]: parallel_result = dview.map_sync(lambda x: x*10, range(32))
```

```
In [67]: serial_result==parallel_result
Out[67]: True
```

---

**Note:** The `DirectView`'s version of `map()` does not do dynamic load balancing. For a load balanced version, use a `LoadBalancedView`.

---

#### See Also:

`map()` is implemented via `ParallelFunction`.

#### Remote function decorators

Remote functions are just like normal functions, but when they are called, they execute on one or more engines, rather than locally. IPython provides two decorators:

```
In [10]: @dview.remote(block=True)
...: def getpid():
...:     import os
...:     return os.getpid()
...:

In [11]: getpid()
Out[11]: [12345, 12346, 12347, 12348]
```

The `@parallel` decorator creates parallel functions, that break up an element-wise operations and distribute them, reconstructing the result.

```

In [12]: import numpy as np

In [13]: A = np.random.random((64,48))

In [14]: @dview.parallel(block=True)
...: def pmul(A,B):
...:     return A*B

In [15]: C_local = A*A

In [16]: C_remote = pmul(A,A)

In [17]: (C_local == C_remote).all()
Out[17]: True

```

**See Also:**

See the docstrings for the `parallel()` and `remote()` decorators for options.

### 5.3.4 Calling Python functions

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

#### apply

The main method for doing remote execution (in fact, all methods that communicate with the engines are built on top of it), is `View.apply()`.

We strive to provide the cleanest interface we can, so *apply* has the following signature:

```
view.apply(f, *args, **kwargs)
```

There are various ways to call functions with IPython, and these flags are set as attributes of the View. The `DirectView` has just two of these flags:

**dv.block** [bool] whether to wait for the result, or return an `AsyncResult` object immediately

**dv.track** [bool] whether to instruct pyzmq to track when This is primarily useful for non-copying sends of numpy arrays that you plan to edit in-place. You need to know when it becomes safe to edit the buffer without corrupting the message.

Creating a view is simple: index-access on a client creates a `DirectView`.

```

In [4]: view = rc[1:3]
Out[4]: <DirectView [1, 2]>

In [5]: view.apply<tab>
view.apply  view.apply_async  view.apply_sync

```

For convenience, you can set block temporarily for a single call with the extra sync/async methods.

## Blocking execution

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

```
In [2]: dview = rc[:] # A DirectView of all engines
In [3]: dview.block=True
In [4]: dview['a'] = 5

In [5]: dview['b'] = 10

In [6]: dview.apply(lambda x: a+b+x, 27)
Out[6]: [42, 42, 42, 42]
```

You can also select blocking execution on a call-by-call basis with the `apply_sync()` method:

```
In [7]: dview.block=False

In [8]: dview.apply_sync(lambda x: a+b+x, 27) Out[8]: [42, 42, 42, 42]
```

Python commands can be executed as strings on specific engines by using a View's `execute` method:

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

In [7]: rc[1:,2].execute('c=a-b')

In [8]: dview['c'] # shorthand for dview.pull('c', block=True)
Out[8]: [15, -5, 15, -5]
```

## Non-blocking execution

In non-blocking mode, `apply()` submits the command to be executed and then returns a `AsyncResult` object immediately. The `AsyncResult` object gives you a way of getting a result at a later time through its `get()` method.

---

**Note:** The `AsyncResult` object provides a superset of the interface in `multiprocessing.pool.AsyncResult`. See the [official Python documentation](#) for more.

---

This allows you to quickly submit long running commands without blocking your local Python/IPython session:

```
# define our function
In [6]: def wait(t):
...:     import time
...:     tic = time.time()
...:     time.sleep(t)
...:     return time.time()-tic
```



```
# In non-blocking mode
In [7]: ar = dview.apply_async(wait, 2)

# Now block for the result
In [8]: ar.get()
Out[8]: [2.0006198883056641, 1.9997570514678955, 1.9996809959411621, 2.0003249645233154]

# Again in non-blocking mode
In [9]: ar = dview.apply_async(wait, 10)

# Poll to see if the result is ready
In [10]: ar.ready()
Out[10]: False

# ask for the result, but wait a maximum of 1 second:
In [45]: ar.get(1)
-----
TimeoutError                                Traceback (most recent call last)
/home/you/<ipython-input-45-7cd858bbb8e0> in <module>()
----> 1 ar.get(1)

/path/to/site-packages/IPython/parallel/asyncreult.pyc in get(self, timeout)
     62         raise self._exception
     63     else:
----> 64         raise error.TimeoutError("Result not ready.")
     65
     66     def ready(self):

TimeoutError: Result not ready.
```

---

**Note:** Note the import inside the function. This is a common model, to ensure that the appropriate modules are imported where the task is run. You can also manually import modules into the engine(s) namespace(s) via `view.execute('import numpy')()`.

---

Often, it is desirable to wait until a set of `AsyncResult` objects are done. For this, there is a the method `wait()`. This method takes a tuple of `AsyncResult` objects (or `msg_ids` or indices to the client's History), and blocks until all of the associated results are ready:

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

# A trivial list of AsyncResults objects
In [73]: pr_list = [dview.apply_async(wait, 3) for i in range(10)]

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

# Then, their results are ready using get() or the '.r' attribute
In [75]: pr_list[0].get()
Out[75]: [2.9982571601867676, 2.9982588291168213, 2.9987530708312988, 2.9990990161895752]
```

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

Most `DirectView` methods (excluding `apply()` and `map()`) 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 `View` 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 for the duration of a single call.

The following examples demonstrate how to use the instance attributes:

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

In [17]: dview.block = False

In [18]: ar = dview.apply(lambda : 10)

In [19]: ar.get()
Out[19]: [10, 10]

In [16]: dview.targets = v.client.ids # all engines (4)

In [21]: dview.block = True

In [22]: dview.apply(lambda : 42)
Out[22]: [42, 42, 42, 42]
```

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

## Parallel magic commands

**Warning:** The magics have not been changed to work with the zeromq system. The magics do work, but *do not* print stdin/out like they used to in `IPython.kernel`.

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()` of the `DirectView`. The `%px` magic executes a single Python command on the engines specified by the `targets` attribute of the `DirectView` instance:

```
# load the parallel magic extension:
In [21]: %load_ext parallelmagic

# Create a DirectView for all targets
In [22]: dv = rc[:]

# Make this DirectView active for parallel magic commands
In [23]: dv.activate()
```

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

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

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

```
Parallel execution on engines: [0, 1, 2, 3]
```

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

```
Parallel execution on engines: [0, 1, 2, 3]
```

```
In [28]: %px ev = numpy.linalg.eigvals(a)
```

```
Parallel execution on engines: [0, 1, 2, 3]
```

```
In [28]: dv['ev']
```

```
Out[28]: [ array([ 1.09522024, -0.09645227]),
          array([ 1.21435496, -0.35546712]),
          array([ 0.72180653,  0.07133042]),
          array([ 1.46384341e+00,  1.04353244e-04])
        ]
```

The `%result` magic gets the most recent result, or takes an argument specifying the index of the result to be requested. It is simply a shortcut to the `get_result()` method:

```
In [29]: dv.apply_async(lambda : ev)
```

```
In [30]: %result
```

```
Out[30]: [ [ 1.28167017  0.14197338],
          [-0.14093616  1.27877273],
          [-0.37023573  1.06779409],
          [ 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]: dv.block=False
```

```
In [31]: %autopx
```

```
Auto Parallel Enabled
```

```
Type %autopx to disable
```

```
In [32]: max_evals = []
```

```
<IPython.parallel.AsyncResult 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.parallel.AsyncResult object at 0x17af8f0>
```

```
In [34]: %autopx
```

Auto Parallel Disabled

```
In [35]: dv.block=True
```

```
In [36]: px ans= "Average max eigenvalue is: %f"%(sum(max_evals)/len(max_evals))
Parallel execution on engines: [0, 1, 2, 3]
```

```
In [37]: dv['ans']
```

```
Out[37]: [ 'Average max eigenvalue is:  10.1387247332',
          'Average max eigenvalue is:  10.2076902286',
          'Average max eigenvalue is:  10.1891484655',
          'Average max eigenvalue is:  10.1158837784',]
```

### 5.3.5 Moving Python objects around

In addition to calling functions and 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]: dview.push(dict(a=1.03234,b=3453))
```

```
Out[38]: [None, None, None, None]
```

```
In [39]: dview.pull('a')
```

```
Out[39]: [ 1.03234, 1.03234, 1.03234, 1.03234]
```

```
In [40]: dview.pull('b', targets=0)
```

```
Out[40]: 3453
```

```
In [41]: dview.pull(('a','b'))
```

```
Out[41]: [ [1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453] ]
```

```
In [43]: dview.push(dict(c='speed'))
```

```
Out[43]: [None, None, None, None]
```

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

```
In [48]: ar = dview.pull('a', block=False)
```

```
In [49]: ar.get()
```

```
Out[49]: [1.03234, 1.03234, 1.03234, 1.03234]
```

#### Dictionary interface

Since a Python namespace is just a `dict`, `DirectView` objects provide dictionary-style access by key and methods such as `get()` and `update()` for convenience. This make the remote namespaces of the

engines appear as a local dictionary. Underneath, these methods call `apply()`:

```
In [51]: dview['a']=['foo','bar']
```

```
In [52]: dview['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 known as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `Client` 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]: dview.scatter('a',range(16))
```

```
Out[58]: [None, None, None, None]
```

```
In [59]: dview['a']
```

```
Out[59]: [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15] ]
```

```
In [60]: dview.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]: dview.scatter('x',range(64))
```

```
In [67]: %px y = [i**10 for i in x]
```

```
Parallel execution on engines: [0, 1, 2, 3]
```

```
Out[67]:
```

```
In [68]: y = dview.gather('y')
```

```
In [69]: print y
```

```
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, ...]
```

## Remote imports

Sometimes you will want to import packages both in your interactive session and on your remote engines. This can be done with the `ContextManager` created by a `DirectView`'s `sync_imports()` method:

```
In [69]: with dview.sync_imports():
...:     import numpy
importing numpy on engine(s)
```

Any imports made inside the block will also be performed on the view's engines. `sync_imports` also takes a *local* boolean flag that defaults to `True`, which specifies whether the local imports should also be performed. However, support for *local=False* has not been implemented, so only packages that can be imported locally will work this way.

You can also specify imports via the `@require` decorator. This is a decorator designed for use in Dependencies, but can be used to handle remote imports as well. Modules or module names passed to `@require` will be imported before the decorated function is called. If they cannot be imported, the decorated function will never execute, and will fail with an `UnmetDependencyError`.

```
In [69]: from IPython.parallel import require
```

```
In [70]: @require('re'):
...: def findall(pat, x):
...:     # re is guaranteed to be available
...:     return re.findall(pat, x)
```

*# you can also pass modules themselves, that you already have locally:*

```
In [71]: @require(time):
...: def wait(t):
...:     time.sleep(t)
...:     return t
```

## 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, we have 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]: dview.block=True
```

```
In [77]: dview.execute('1/0')
```

```
-----
CompositeError                                Traceback (most recent call last)
/home/you/<ipython-input-10-15c2c22dec39> in <module>()
----> 1 dview.execute('1/0', block=True)

/path/to/site-packages/IPython/parallel/view.py in execute(self, code, block)
    460         default: self.block
    461         """
--> 462         return self.apply_with_flags(util._execute, args=(code,), block=block)
    463
    464         def run(self, filename, block=None):

/home/you/<string> in apply_with_flags(self, f, args, kwargs, block, track)
```

```

/path/to/site-packages/IPython/parallel/view.py in sync_results(f, self, *args, **kwargs)
    46 def sync_results(f, self, *args, **kwargs):
    47     """sync relevant results from self.client to our results attribute."""
--> 48     ret = f(self, *args, **kwargs)
    49     delta = self.outstanding.difference(self.client.outstanding)
    50     completed = self.outstanding.intersection(delta)

/home/you/<string> in apply_with_flags(self, f, args, kwargs, block, track)

/path/to/site-packages/IPython/parallel/view.py in save_ids(f, self, *args, **kwargs)
    35     n_previous = len(self.client.history)
    36     try:
--> 37         ret = f(self, *args, **kwargs)
    38     finally:
    39         nmsgs = len(self.client.history) - n_previous

/path/to/site-packages/IPython/parallel/view.py in apply_with_flags(self, f, args, kwargs,
    398         if block:
    399             try:
--> 400                 return ar.get()
    401             except KeyboardInterrupt:
    402                 pass

/path/to/site-packages/IPython/parallel/asyncresult.pyc in get(self, timeout)
    87         return self._result
    88     else:
--> 89         raise self._exception
    90     else:
    91         raise error.TimeoutError("Result not ready.")

```

```

CompositeError: one or more exceptions from call to method: _execute
[0:apply]: ZeroDivisionError: integer division or modulo by zero
[1:apply]: ZeroDivisionError: integer division or modulo by zero
[2:apply]: ZeroDivisionError: integer division or modulo by zero
[3:apply]: 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:
.....:     dview.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.pyc 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]: dview.execute('1/0')
-----
CompositeError                                Traceback (most recent call last)
/home/you/<ipython-input-10-15c2c22dec39> in <module>()
----> 1 dview.execute('1/0', block=True)

/path/to/site-packages/IPython/parallel/view.py in execute(self, code, block)
    460         default: self.block
    461         """
--> 462         return self.apply_with_flags(util._execute, args=(code,), block=block)
    463
    464         def run(self, filename, block=None):

/home/you/<string> in apply_with_flags(self, f, args, kwargs, block, track)

/path/to/site-packages/IPython/parallel/view.py in sync_results(f, self, *args, **kwargs)
    46 def sync_results(f, self, *args, **kwargs):
    47     """sync relevant results from self.client to our results attribute."""
--> 48     ret = f(self, *args, **kwargs)
    49     delta = self.outstanding.difference(self.client.outstanding)
    50     completed = self.outstanding.intersection(delta)

/home/you/<string> in apply_with_flags(self, f, args, kwargs, block, track)

/path/to/site-packages/IPython/parallel/view.py in save_ids(f, self, *args, **kwargs)
    35     n_previous = len(self.client.history)
    36     try:
--> 37         ret = f(self, *args, **kwargs)
    38     finally:
    39         nmsgs = len(self.client.history) - n_previous

/path/to/site-packages/IPython/parallel/view.py in apply_with_flags(self, f, args, kwargs,
    398         if block:
    399             try:
--> 400                 return ar.get()
    401             except KeyboardInterrupt:
    402                 pass

/path/to/site-packages/IPython/parallel/asyncresult.pyc in get(self, timeout)
    87         return self._result
    88         else:
--> 89         raise self._exception
    90     else:
```



```

91             raise error.TimeoutError("Result not ready.")

CompositeError: one or more exceptions from call to method: _execute
[0:apply]: ZeroDivisionError: integer division or modulo by zero
[1:apply]: ZeroDivisionError: integer division or modulo by zero
[2:apply]: ZeroDivisionError: integer division or modulo by zero
[3:apply]: ZeroDivisionError: integer division or modulo by zero

In [82]: %debug
> /path/to/site-packages/IPython/parallel/asyncreult.py(80)get()
   79             else:
--> 80                 raise self._exception
   81             else:

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:apply]:
Traceback (most recent call last):
  File "/path/to/site-packages/IPython/parallel/streamkernel.py", line 332, in apply_request
    exec code in working, working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/client.py", line 69, in _execute
    exec code in globals()
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

[1:apply]:
Traceback (most recent call last):
  File "/path/to/site-packages/IPython/parallel/streamkernel.py", line 332, in apply_request
    exec code in working, working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/client.py", line 69, in _execute
    exec code in globals()
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

[2:apply]:
Traceback (most recent call last):
  File "/path/to/site-packages/IPython/parallel/streamkernel.py", line 332, in apply_request
    exec code in working, working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/client.py", line 69, in _execute
    exec code in globals()
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

```

```
[3:apply]:
Traceback (most recent call last):
  File "/path/to/site-packages/IPython/parallel/streamkernel.py", line 332, in apply_request
    exec code in working, working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/client.py", line 69, in _execute
    exec code in globals()
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

---

**Note:** TODO: The above tracebacks are not up to date

---

All of this same error handling magic even works in non-blocking mode:

```
In [83]: dview.block=False
```

```
In [84]: ar = dview.execute('1/0')
```

```
In [85]: ar.get()
```

```
-----
CompositeError                                Traceback (most recent call last)
/Users/minrk/<ipython-input-3-8531eb3d26fb> in <module>()
----> 1 ar.get()

/path/to/site-packages/IPython/parallel/asyncresult.pyc in get(self, timeout)
    78         return self._result
    79     else:
----> 80         raise self._exception
    81     else:
    82         raise error.TimeoutError("Result not ready.")

CompositeError: one or more exceptions from call to method: _execute
[0:apply]: ZeroDivisionError: integer division or modulo by zero
[1:apply]: ZeroDivisionError: integer division or modulo by zero
[2:apply]: ZeroDivisionError: integer division or modulo by zero
[3:apply]: ZeroDivisionError: integer division or modulo by zero
```

## 5.4 The IPython task interface

The task interface to the cluster presents the engines as a fault tolerant, dynamic load-balanced system of workers. Unlike the multiengine interface, in the task interface the user have no direct access to individual engines. By allowing the IPython scheduler to assign work, this interface is simultaneously simpler and more powerful.

Best of all, the user can use both of these interfaces running at the same time to take advantage of their respective 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 start -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 Client instance

The first step is to import the IPython `IPython.parallel` module and then create a `Client` instance, and we will also be using a `LoadBalancedView`, here called *lview*:

```
In [1]: from IPython.parallel import Client
```

```
In [2]: rc = Client()
```

This form assumes that the controller was started on localhost with default configuration. If not, the location of the controller must be given as an argument to the constructor:

```
# for a visible LAN controller listening on an external port:
```

```
In [2]: rc = Client('tcp://192.168.1.16:10101')
```

```
# or to connect with a specific profile you have set up:
```

```
In [3]: rc = Client(profile='mpi')
```

For load-balanced execution, we will make use of a `LoadBalancedView` object, which can be constructed via the client's `load_balanced_view()` method:

```
In [4]: lview = rc.load_balanced_view() # default load-balanced view
```

#### See Also:

For more information, see the in-depth explanation of [Views](#).

### 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, these can be implemented via the task interface. The exact same tools can perform these actions in load-balanced ways as well as multiplexed ways: a parallel version of `map()` and `@parallel()` function decorator. If one specifies the argument *balanced=True*, then 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

To load-balance `map()`, simply use a `LoadBalancedView`:

```
In [62]: lview.block = True

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

In [64]: parallel_result = lview.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]: @lview.parallel()
.....: def f(x):
.....:     return 10.0*x**4
.....:

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

## 5.4.4 Dependencies

Often, pure atomic load-balancing is too primitive for your work. In these cases, you may want to associate some kind of *Dependency* that describes when, where, or whether a task can be run. In IPython, we provide two types of dependencies: [Functional Dependencies](#) and [Graph Dependencies](#)

---

**Note:** It is important to note that the pure ZeroMQ scheduler does not support dependencies, and you will see errors or warnings if you try to use dependencies with the pure scheduler.

---

## Functional Dependencies

Functional dependencies are used to determine whether a given engine is capable of running a particular task. This is implemented via a special `Exception` class, `UnmetDependency`, found in `IPython.parallel.error`. Its use is very simple: if a task fails with an `UnmetDependency` exception, then the scheduler, instead of relaying the error up to the client like any other error, catches the error, and submits the task to a different engine. This will repeat indefinitely, and a task will never be submitted to a given engine a second time.

You can manually raise the `UnmetDependency` yourself, but IPython has provided some decorators for facilitating this behavior.

There are two decorators and a class used for functional dependencies:

```
In [9]: from IPython.parallel import depend, require, dependent
```

## @require

The simplest sort of dependency is requiring that a Python module is available. The `@require` decorator lets you define a function that will only run on engines where names you specify are importable:

```
In [10]: @require('numpy', 'zmq')
...: def myfunc():
...:     return dostuff()
```

Now, any time you apply `myfunc()`, the task will only run on a machine that has `numpy` and `pyzmq` available, and when `myfunc()` is called, `numpy` and `zmq` will be imported.

## @depend

The `@depend` decorator lets you decorate any function with any *other* function to evaluate the dependency. The dependency function will be called at the start of the task, and if it returns `False`, then the dependency will be considered unmet, and the task will be assigned to another engine. If the dependency returns *anything other than* `False`, the rest of the task will continue.

```
In [10]: def platform_specific(plat):
...:     import sys
...:     return sys.platform == plat

In [11]: @depend(platform_specific, 'darwin')
...: def mactask():
...:     do_mac_stuff()

In [12]: @depend(platform_specific, 'nt')
...: def wintask():
...:     do_windows_stuff()
```

In this case, any time you apply `mytask`, it will only run on an OSX machine. `@depend` is just like `apply`, in that it has a `@depend(f, *args, **kwargs)` signature.

## dependents

You don't have to use the decorators on your tasks, if for instance you may want to run tasks with a single function but varying dependencies, you can directly construct the `dependent` object that the decorators use:

## Graph Dependencies

Sometimes you want to restrict the time and/or location to run a given task as a function of the time and/or location of other tasks. This is implemented via a subclass of `set`, called a `Dependency`. A `Dependency` is just a set of `msg_ids` corresponding to tasks, and a few attributes to guide how to decide when the `Dependency` has been met.

The switches we provide for interpreting whether a given dependency set has been met:

**any|all** Whether the dependency is considered met if *any* of the dependencies are done, or only after *all* of them have finished. This is set by a Dependency's `all` boolean attribute, which defaults to `True`.

**success [default: True]** Whether to consider tasks that succeeded as fulfilling dependencies.

**failure [default [False]]** Whether to consider tasks that failed as fulfilling dependencies. using *failure=True, success=False* is useful for setting up cleanup tasks, to be run only when tasks have failed.

Sometimes you want to run a task after another, but only if that task succeeded. In this case, `success` should be `True` and `failure` should be `False`. However sometimes you may not care whether the task succeeds, and always want the second task to run, in which case you should use *success=failure=True*. The default behavior is to only use successes.

There are other switches for interpretation that are made at the *task* level. These are specified via keyword arguments to the client's `apply()` method.

**after, follow** You may want to run a task *after* a given set of dependencies have been run and/or run it *where* another set of dependencies are met. To support this, every task has an *after* dependency to restrict time, and a *follow* dependency to restrict destination.

**timeout** You may also want to set a time-limit for how long the scheduler should wait before a task's dependencies are met. This is done via a *timeout*, which defaults to 0, which indicates that the task should never timeout. If the timeout is reached, and the scheduler still hasn't been able to assign the task to an engine, the task will fail with a `DependencyTimeout`.

---

**Note:** Dependencies only work within the task scheduler. You cannot instruct a load-balanced task to run after a job submitted via the MUX interface.

---

The simplest form of Dependencies is with *all=True, success=True, failure=False*. In these cases, you can skip using Dependency objects, and just pass `msg_ids` or `AsyncResult` objects as the *follow* and *after* keywords to `client.apply()`:

```
In [14]: client.block=False
```

```
In [15]: ar = lview.apply(f, args, kwargs)
```

```
In [16]: ar2 = lview.apply(f2)
```

```
In [17]: ar3 = lview.apply_with_flags(f3, after=[ar, ar2])
```

```
In [17]: ar4 = lview.apply_with_flags(f3, follow=[ar], timeout=2.5)
```

### See Also:

Some parallel workloads can be described as a [Directed Acyclic Graph](#), or DAG. See [DAG Dependencies](#) for an example demonstrating how to use map a NetworkX DAG onto task dependencies.

### Impossible Dependencies

The schedulers do perform some analysis on graph dependencies to determine whether they are not possible to be met. If the scheduler does discover that a dependency cannot be met, then the task will fail with an

ImpossibleDependency error. This way, if the scheduler realized that a task can never be run, it won't sit indefinitely in the scheduler clogging the pipeline.

The basic cases that are checked:

- depending on nonexistent messages
- *follow* dependencies were run on more than one machine and *all=True*
- any dependencies failed and *all=True, success=True, failures=False*
- all dependencies failed and *all=False, success=True, failure=False*

**Warning:** This analysis has not been proven to be rigorous, so it is likely possible for tasks to become impossible to run in obscure situations, so a timeout may be a good choice.

### 5.4.5 Schedulers

There are a variety of valid ways to determine where jobs should be assigned in a load-balancing situation. In IPython, we support several standard schemes, and even make it easy to define your own. The scheme can be selected via the `--scheme` argument to **ipcontroller**, or in the `HubFactory.scheme` attribute of a controller config object.

The built-in routing schemes:

To select one of these schemes, simply do:

```
$ ipcontroller --scheme <schemename>
for instance:
$ ipcontroller --scheme lru
```

**lru:** Least Recently Used

Always assign work to the least-recently-used engine. A close relative of round-robin, it will be fair with respect to the number of tasks, agnostic with respect to runtime of each task.

**plainrandom:** Plain Random

Randomly picks an engine on which to run.

**twobin:** Two-Bin Random

**Requires numpy**

Pick two engines at random, and use the LRU of the two. This is known to be better than plain random in many cases, but requires a small amount of computation.

**leastload:** Least Load

**This is the default scheme**

Always assign tasks to the engine with the fewest outstanding tasks (LRU breaks tie).

**weighted:** Weighted Two-Bin Random

### Requires numpy

Pick two engines at random using the number of outstanding tasks as inverse weights, and use the one with the lower load.

## Pure ZMQ Scheduler

For maximum throughput, the ‘pure’ scheme is not Python at all, but a C-level `MonitoredQueue` from `PyZMQ`, which uses a ZeroMQ `XREQ` socket to perform all load-balancing. This scheduler does not support any of the advanced features of the `Python Scheduler`.

Disabled features when using the ZMQ Scheduler:

- **Engine unregistration** Task farming will be disabled if an engine unregisters. Further, if an engine is unregistered during computation, the scheduler may not recover.
- **Dependencies** Since there is no Python logic inside the Scheduler, routing decisions cannot be made based on message content.
- **Early destination notification** The Python schedulers know which engine gets which task, and notify the Hub. This allows graceful handling of Engines coming and going. There is no way to know where ZeroMQ messages have gone, so there is no way to know what tasks are on which engine until they *finish*. This makes recovery from engine shutdown very difficult.

---

**Note:** TODO: performance comparisons

---

## 5.4.6 More details

The `LoadBalancedView` 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:

- `LoadBalancedView`
- `AsyncResult`
- `apply()`
- `dependency`

The following is an overview of how to use these classes together:

1. Create a `Client` and `LoadBalancedView`
2. Define some functions to be run as tasks
3. Submit your tasks to using the `apply()` method of your `LoadBalancedView` instance.
4. Use `Client.get_result()` to get the results of the tasks, or use the `AsyncResult.get()` method of the results to wait for and then receive the results.

**See Also:**

A demo of *DAG Dependencies* with `NetworkX` and `IPython`.



## 5.5 Using MPI with IPython

---

**Note:** Not adapted to zmq yet This is out of date wrt ipcluster in general as well

---

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:

```
$ ipcluster start -p mpi -n 4
```

---

**Note:** It is assumed here that the mpi profile has been set up, as described [here](#).

---

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.parallel import Client
```

```
In [2]: %load_ext parallel_magic
```

```
In [3]: c = Client(profile='mpi')

In [4]: view = c[:]

In [5]: view.activate()

# run the contents of the file on each engine:
In [6]: view.run('psum.py')

In [6]: px a = np.random.rand(100)
Parallel execution on engines: [0,1,2,3]

In [8]: px s = psum(a)
Parallel execution on engines: [0,1,2,3]

In [9]: view['s']
Out[9]: [187.451545803, 187.451545803, 187.451545803, 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

---

**Note:** This section is not thorough, and IPython.zmq needs a thorough security audit.

---

IPython's `IPython.zmq` 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 *hub*. This process monitors a set of engines and schedulers, and keeps track of the state of the processes. It listens for registration connections from engines and clients, and monitor connections from schedulers.
- The IPython *schedulers*. This is a set of processes that relay commands and results between clients and engines. They are typically on the same machine as the controller, and listen for connections from engines and clients, but connect to the Hub.
- 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 processes are called the IPython *kernel*, and the hub and schedulers together are referred to as the *controller*.

---

**Note:** Are these really still referred to as the Kernel? It doesn't seem so to me. 'cluster' seems more accurate.

-MinRK

---

These processes communicate over any transport supported by ZeroMQ (tcp,pgm,infiniband,ipc) with a well defined topology. The IPython hub and schedulers listen on sockets. Upon starting, an engine connects to a hub and registers itself, which then informs the engine of the connection information for the schedulers, and the engine then connects to the schedulers. These engine/hub and engine/scheduler connections persist for the lifetime of each engine.

The IPython client also connects to the controller processes using a number of socket connections. As of writing, this is one socket per scheduler (4), and 3 connections to the hub for a total of 7. 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 hub, schedulers, 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 schedulers 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

ZeroMQ provides exactly no security. For this reason, users of IPython must be very careful in managing connections, because an open TCP/IP socket presents access to arbitrary execution as the user on the engine machines. As a result, the default behavior of controller processes is to only listen for clients on the loopback interface, and the client must establish SSH tunnels to connect to the controller processes.

**Warning:** If the controller’s loopback interface is untrusted, then IPython should be considered vulnerable, and this extends to the loopback of all connected clients, which have opened a loopback port that is redirected to the controller’s loopback port.

## SSH

Since ZeroMQ provides no security, SSH tunnels are the primary source of secure connections. A connector file, such as *ipcontroller-client.json*, will contain information for connecting to the controller, possibly including the address of an ssh-server through which the client is to tunnel. The Client object then creates tunnels using either [\[OpenSSH\]](#) or [\[Paramiko\]](#), depending on the platform. If users do not wish to use OpenSSH or Paramiko, or the tunneling utilities are insufficient, then they may construct the tunnels themselves, and simply connect clients and engines as if the controller were on loopback on the connecting machine.

---

**Note:** There is not currently tunneling available for engines.

---

## Authentication

To protect users of shared machines, an execution key is used to authenticate all messages.

The Session object that handles the message protocol uses a unique key to verify valid messages. This can be any value specified by the user, but the default behavior is a pseudo-random 128-bit number, as generated by *uuid.uuid4()*. This key is checked on every message everywhere it is unpacked (Controller, Engine, and Client) to ensure that it came from an authentic user, and no messages that do not contain this key are acted upon in any way.

There is exactly one key per cluster - it must be the same everywhere. Typically, the controller creates this key, and stores it in the private connection files *ipython-{engine|client}.json*. These files are typically stored in the *~/.ipython/cluster\_<profile>/security* directory, and are maintained as readable only by the owner, just as is common practice with a user’s keys in their *.ssh* directory.

**Warning:** It is important to note that the key authentication, as emphasized by the use of a uuid rather than generating a key with a cryptographic library, provides a defense against *accidental* messages more than it does against malicious attacks. If loopback is compromised, it would be trivial for an attacker to intercept messages and deduce the key, as there is no encryption.

### 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.

On the first level, this attack is prevented by requiring access to the controller's ports, which are recommended to only be open on loopback if the controller is on an untrusted local network. If the attacker does have access to the Controller's ports, then the attack is prevented by the capabilities based client authentication of the execution key. The relevant authentication information is encoded into the JSON file that clients must present to gain access to the IPython controller. By limiting the distribution of those keys, a user can grant access to only authorized persons, just as with SSH keys.

It is highly unlikely that an execution key 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  $2^{128}$ .

**Warning:** If the attacker has gained enough access to intercept loopback connections on *either* the controller or client, then the key is easily deduced from network traffic.

## 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 capabilities in a connection file, which ensure that a client or engine connects to the correct controller. It is also important to note that the connection files 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 the key to use for that connection. Thus, in order to introduce a hostile controller, the attacker must convince the user to use the key associated with the hostile controller. As long as a user is diligent in only using keys from trusted sources, this attack is not possible.

---

**Note:** I may be wrong, the unauthorized controller may be easier to fake than this.

---

### 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 additional 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 SSH tunneled 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

---

**Note:** this does not apply to ZMQ, but I am sure there will be 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

---

**Note:** Not adapted to zmq yet

---

### 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 2.6 or 2.7 (<http://www.python.org>)
- pywin32 (<http://sourceforge.net/projects/pywin32/>)
- PyReadline (<https://launchpad.net/pyreadline>)
- pyzmq (<http://github.com/zeromq/pyzmq/downloads>)
- 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>)
- 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:\Python27` and `C:\Python27\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:

```

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.

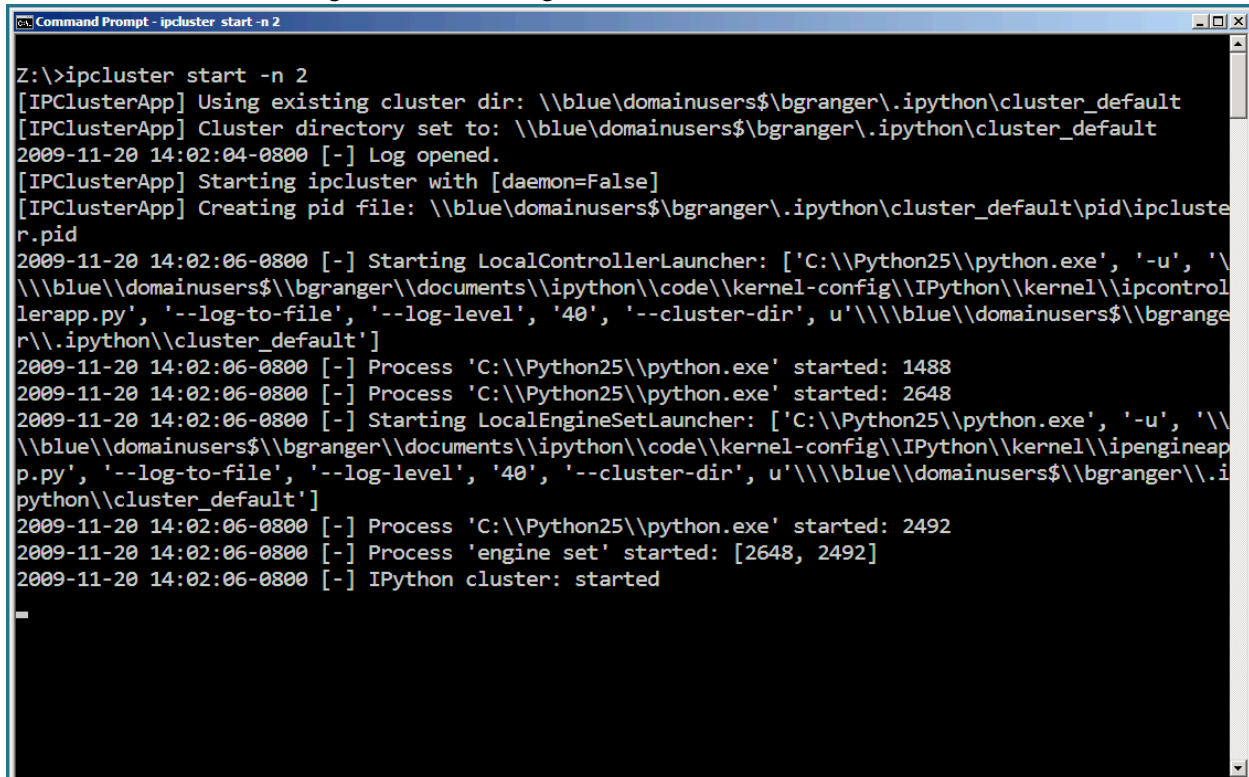
IPython has a command line program called **ipcluster** that automates all aspects of starting the controller and engines on the compute nodes. **ipcluster** has full support for the Windows HPC job scheduler, meaning that **ipcluster** can use this job scheduler to start the controller and engines. 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:



```
Command Prompt - ipcluster start -n 2

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.

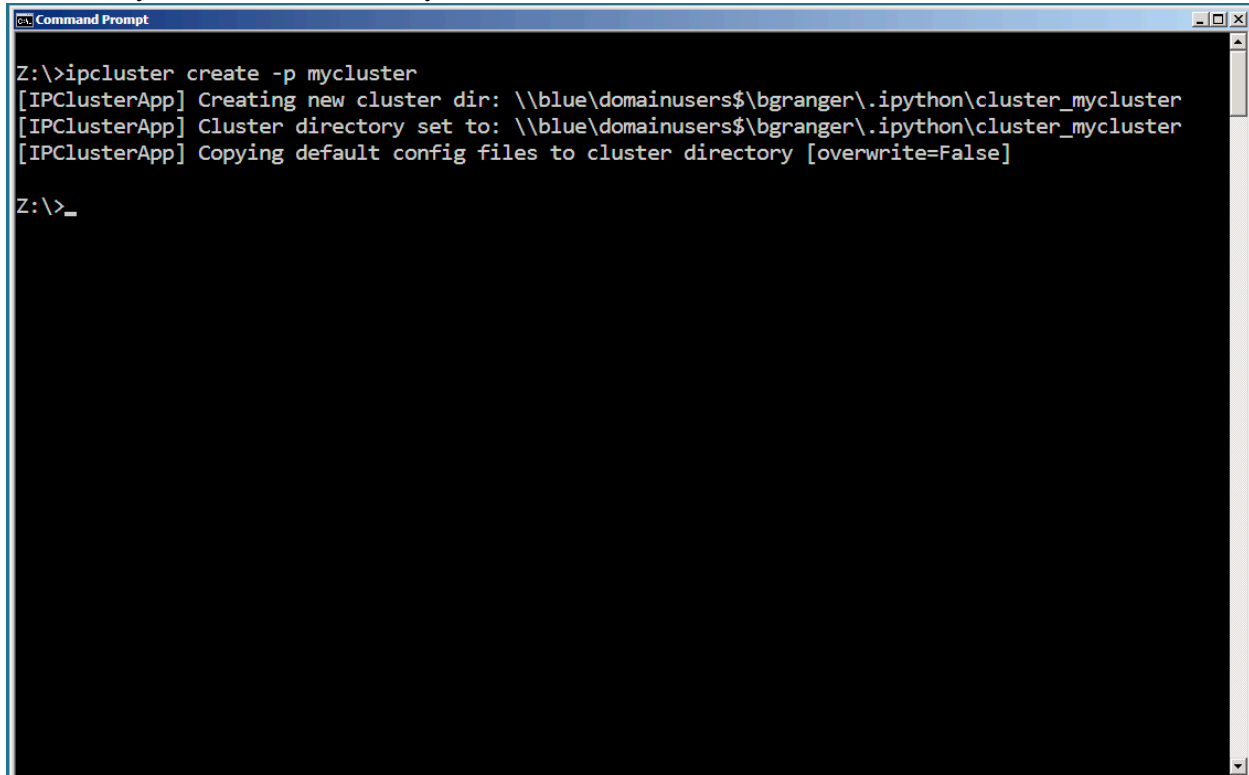
Associated with each cluster profile is a cluster directory. This cluster directory is a specially named directory (typically located in the `.ipython` subdirectory of your home directory) that contains the configura-

tion 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.parallel.apps.launcher.WindowsHPCControllerLauncher'
c.Global.engine_launcher = \
    'IPython.parallel.apps.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
```

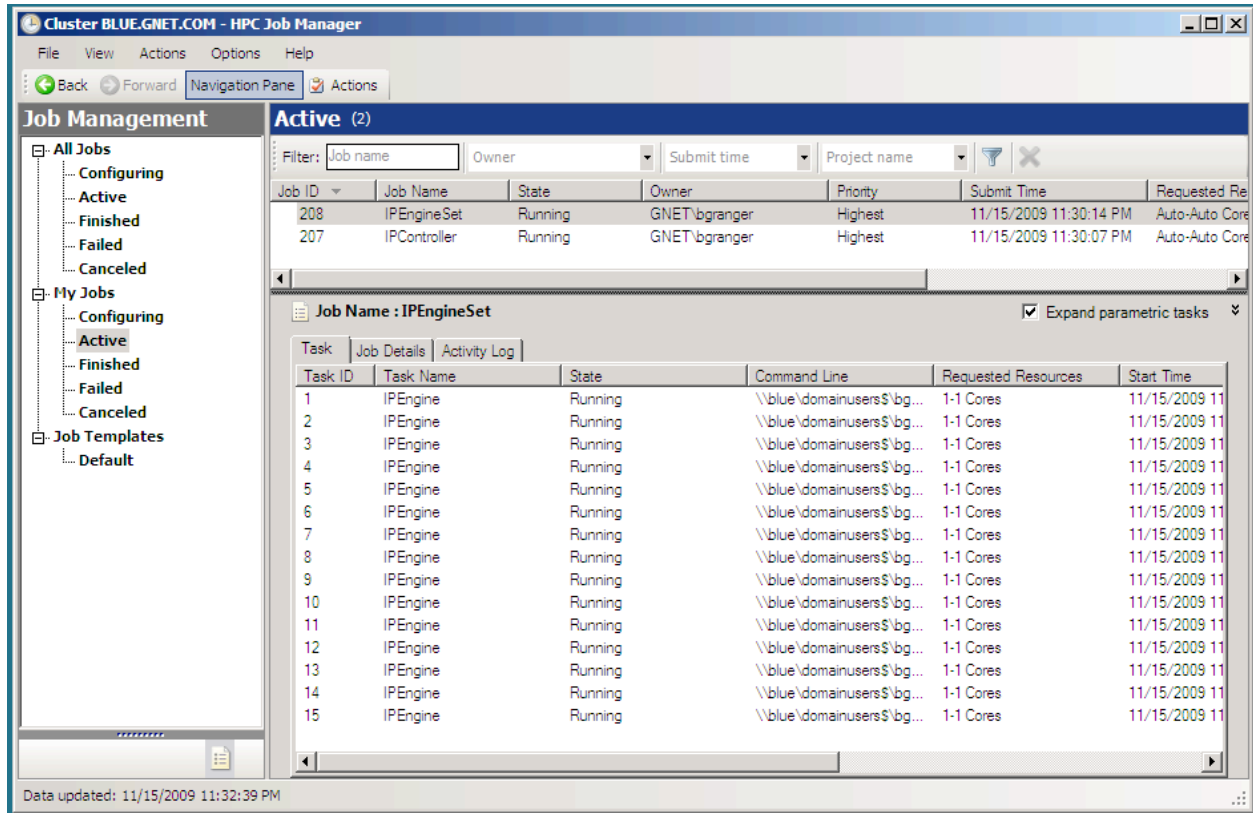
The `-n` option tells **ipcluster** how many engines to start (in this case 32). Stopping the cluster is as simple as typing Control-C.

## 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.parallel import *
```

```
In [2]: c = MultiEngineClient(profile='mycluster')
```

```
In [3]: mec.get_ids()
```

```
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

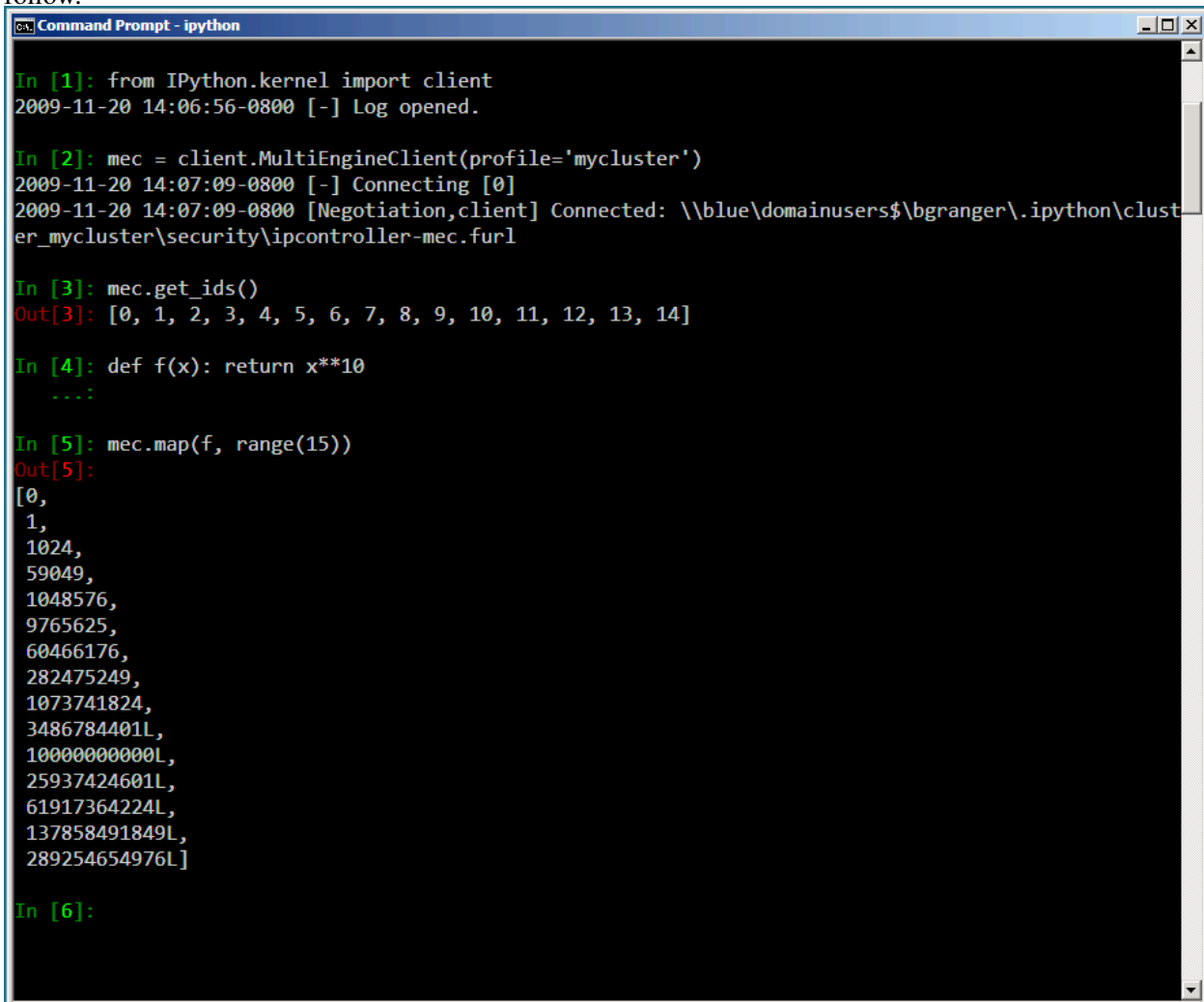
```
In [4]: def f(x):
...:     return x**10
```

```
In [5]: mec.map(f, range(15)) # 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.



```
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

---

**Note:** Performance numbers from `IPython.kernel`, not `newparallel`.

---

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 --pylab
```

at the system command line.

### 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/newparallel` 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` 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
```

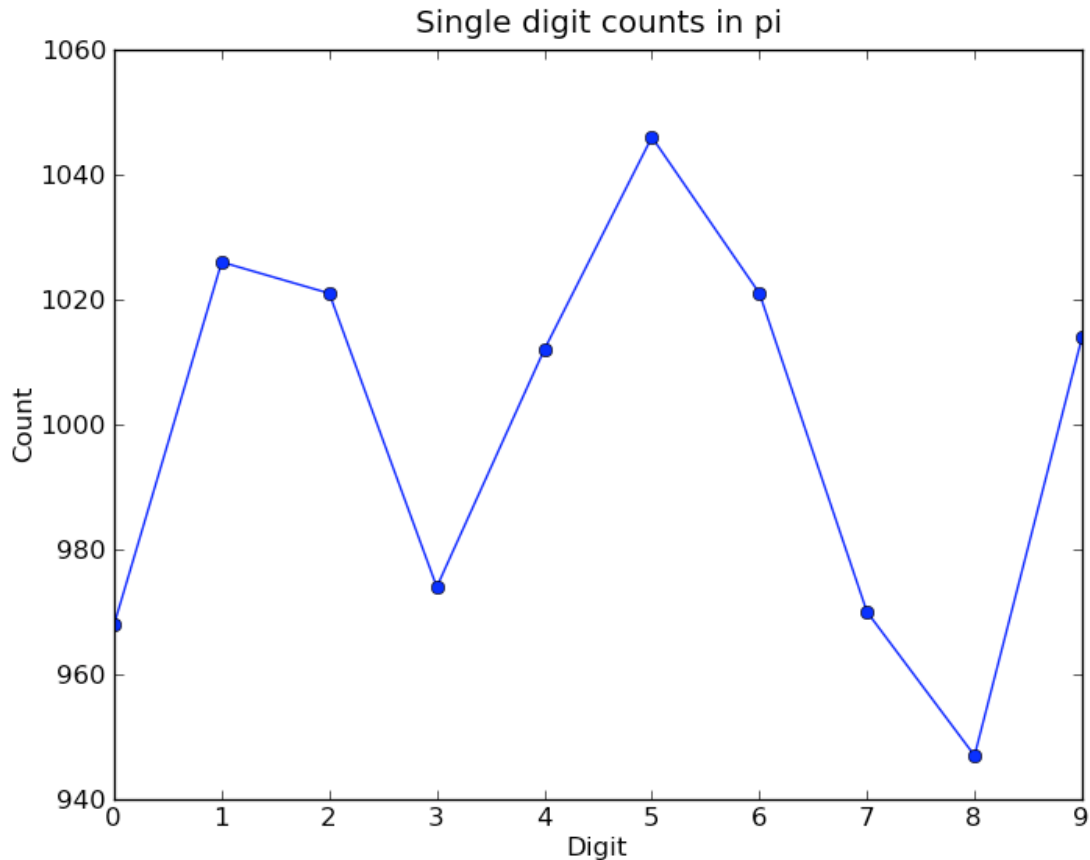
```
In [12]: run pidigits.py # load one_digit_freqs/plot_one_digit_freqs
```

```
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`:

```

    Read digits of pi from a file and compute the 1 digit frequencies.
    """
    d = txt_file_to_digits(filename)
    freqs = one_digit_freqs(d)
    return freqs

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):
    """

```

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/newparallel/parallempi.py`. This code can be run in parallel using IPython by following these steps:

1. 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.
2. With the file `parallempi.py` in your current working directory, open up IPython in pylab mode and type `run parallempi.py`. This will download the pi files via ftp the first time you run it, if they are not present in the Engines' working directory.

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.parallel import Client

# The Client allows us to use the engines interactively.
# We simply pass Client the name of the cluster profile we
# are using.
In [2]: c = Client(profile='mycluster')
In [3]: view = c.load_balanced_view()

In [3]: c.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'

# 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',
 'pi200m.ascii.02of20',
 'pi200m.ascii.03of20',
 'pi200m.ascii.04of20',
 'pi200m.ascii.05of20',
 'pi200m.ascii.06of20',
 'pi200m.ascii.07of20',
 'pi200m.ascii.08of20',
 'pi200m.ascii.09of20',
 'pi200m.ascii.10of20',
 'pi200m.ascii.11of20',
 'pi200m.ascii.12of20',
 'pi200m.ascii.13of20',
 'pi200m.ascii.14of20',
 'pi200m.ascii.15of20']

# download the data files if they don't already exist:
In [8]: v.map(fetch_pi_file, files)

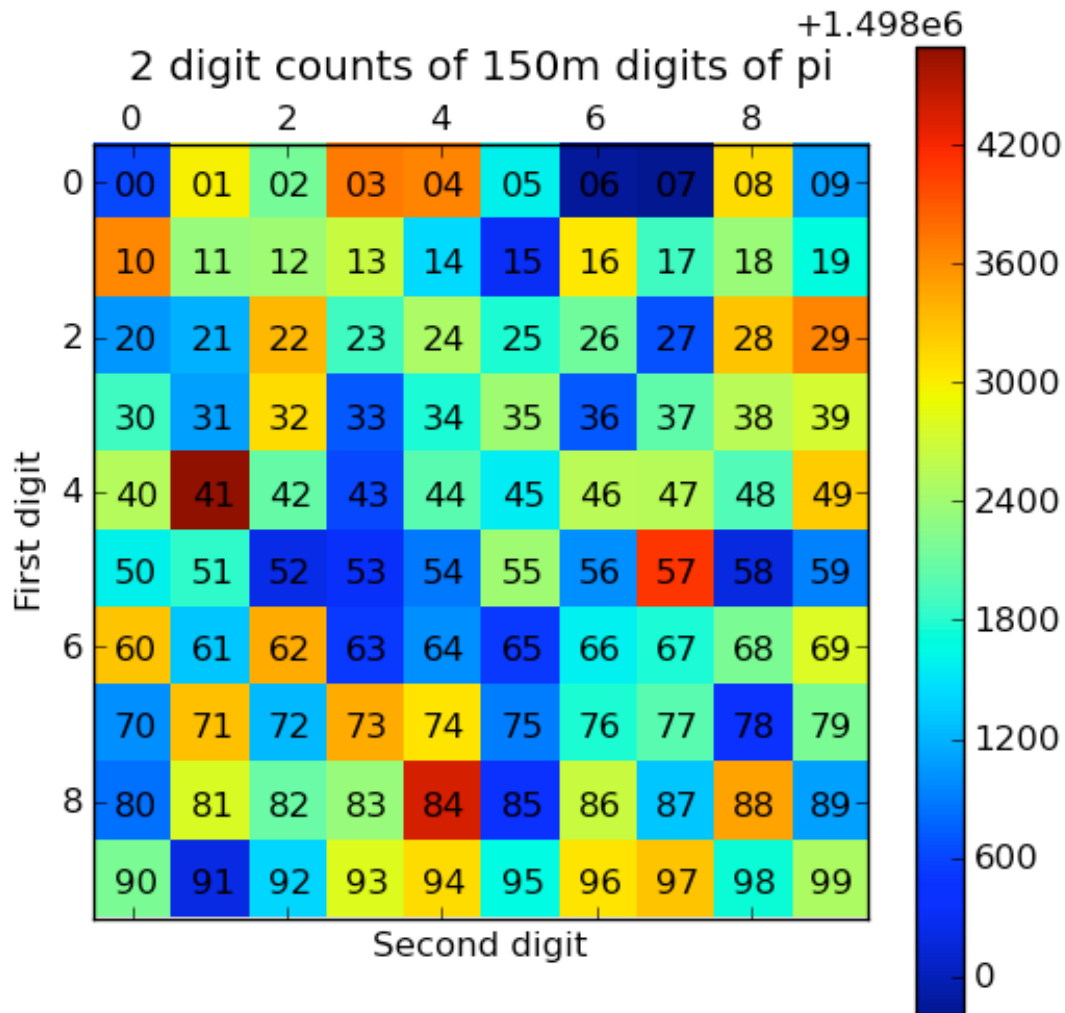
# This is the parallel calculation using the Client.map method
# which applies compute_two_digit_freqs to each file in files in parallel.
In [9]: freqs_all = v.map(compute_two_digit_freqs, files)

# Add up the frequencies from each engine.
In [10]: freqs = reduce_freqs(freqs_all)

In [11]: plot_two_digit_freqs(freqs)
Out[11]: <matplotlib.image.AxesImage object at 0x18beb110>

In [12]: plt.title('2 digit counts of 150m digits of pi')
Out[12]: <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/newparallel 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:

```
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.
    """
    import numpy as np
    from math import exp, sqrt

    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 `LoadBalancedView` class, which distributes work to the engines using dynamic load balancing. This view is a wrapper of the `Client` class shown in the previous example. The parallel calculation using `LoadBalancedView` 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.parallel 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 Client is used to setup the calculation and works with all
# engines.
c = Client(profile=cluster_profile)

# A LoadBalancedView is an interface to the engines that provides dynamic load
# balancing at the expense of not knowing which engine will execute the code.
view = c.load_balanced_view()
```

```
# Initialize the common code on the engines. This Python module has the
# price_options function that prices the options.

#-----
# 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()
async_results = []
for strike in strike_vals:
    for sigma in sigma_vals:
        ar = view.apply_async(price_options, price, strike, sigma, rate, days, paths)
        async_results.append(ar)

print "Submitted tasks: ", len(async_results)
sys.stdout.flush()

# Block until all tasks are completed.
c.wait(async_results)
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 = [ar.get() for ar in async_results]

# 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 in enumerate(results):
    prices[i] = tuple(price)

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:

```

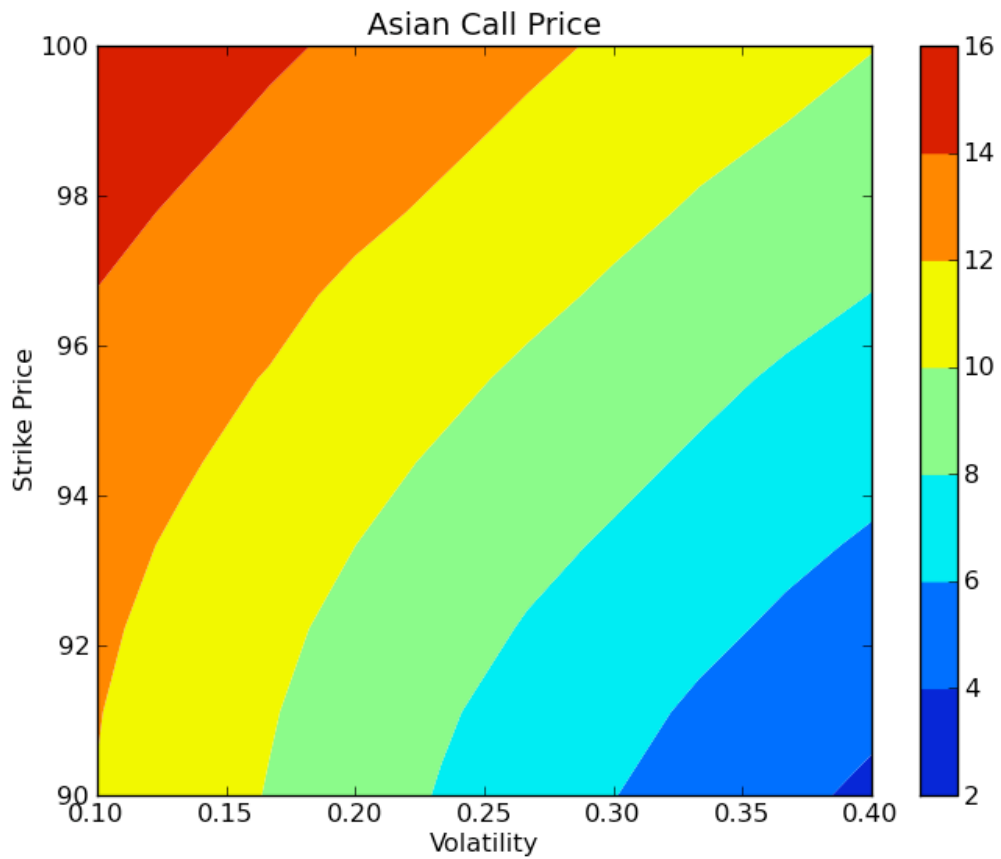
In [8]: plot_options(sigma_vals, K_vals, prices['acall'])

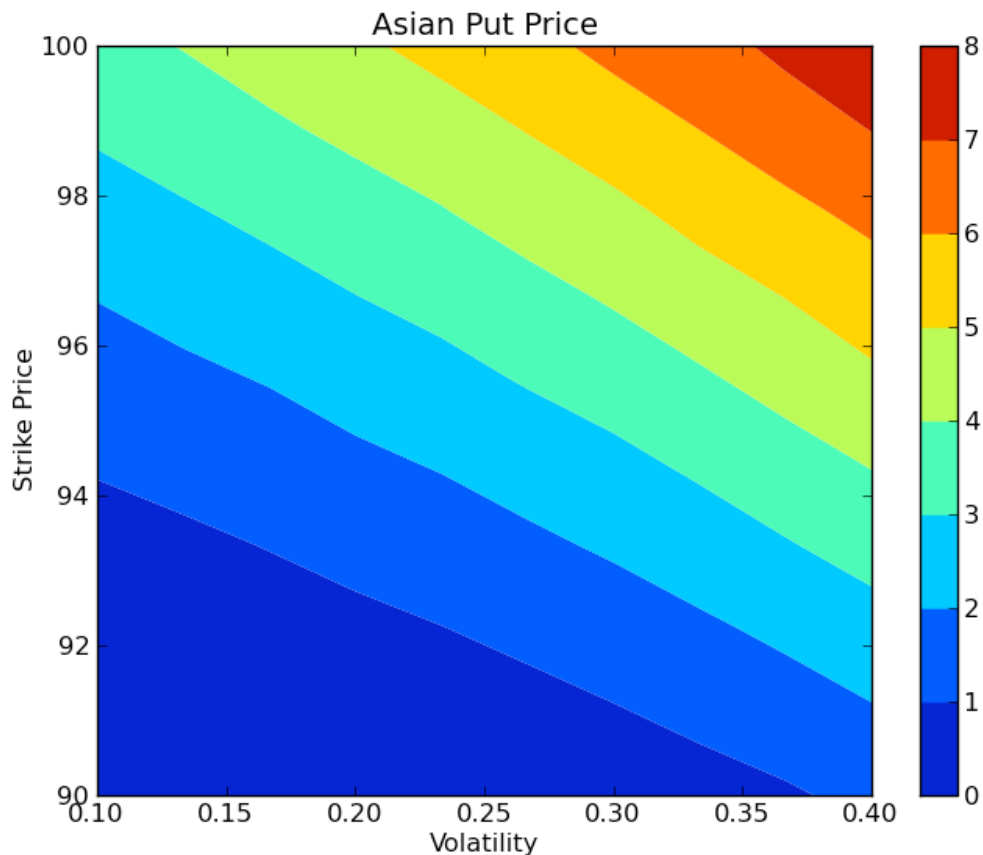
In [9]: plt.figure()
Out[9]: <matplotlib.figure.Figure object at 0x18c178d0>

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 `DirectView` and `LoadBalancedView` 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.

---

**Note:** The newparallel code has never been run on Windows HPC Server, so the last conclusion is untested.

---

## 5.9 DAG Dependencies

Often, parallel workflow is described in terms of a [Directed Acyclic Graph](#) or DAG. A popular library for working with Graphs is [NetworkX](#). Here, we will walk through a demo mapping a nx DAG to task dependencies.

The full script that runs this demo can be found in `docs/examples/newparallel/dagdeps.py`.

### 5.9.1 Why are DAGs good for task dependencies?

The ‘G’ in DAG is ‘Graph’. A Graph is a collection of **nodes** and **edges** that connect the nodes. For our purposes, each node would be a task, and each edge would be a dependency. The ‘D’ in DAG stands for ‘Directed’. This means that each edge has a direction associated with it. So we can interpret the edge (a,b) as meaning that b depends on a, whereas the edge (b,a) would mean a depends on b. The ‘A’ is ‘Acyclic’, meaning that there must not be any closed loops in the graph. This is important for dependencies, because if a loop were closed, then a task could ultimately depend on itself, and never be able to run. If your workflow can be described as a DAG, then it is impossible for your dependencies to cause a deadlock.

### 5.9.2 A Sample DAG

Here, we have a very simple 5-node DAG:

With NetworkX, an arrow is just a fattened bit on the edge. Here, we can see that task 0 depends on nothing, and can run immediately. 1 and 2 depend on 0; 3 depends on 1 and 2; and 4 depends only on 1.

A possible sequence of events for this workflow:

0. Task 0 can run right away
1. 0 finishes, so 1,2 can start
2. 1 finishes, 3 is still waiting on 2, but 4 can start right away
3. 2 finishes, and 3 can finally start

Further, taking failures into account, assuming all dependencies are run with the default *success=True,failure=False*, the following cases would occur for each node’s failure:

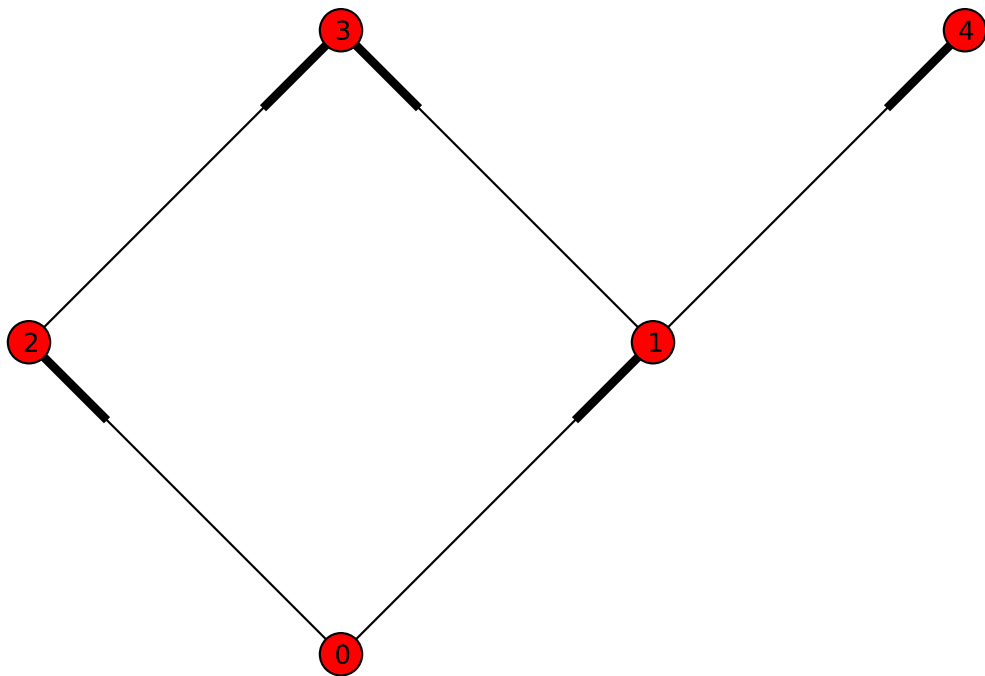
0. fails: all other tasks fail as Impossible
1. 2 can still succeed, but 3,4 are unreachable
2. 3 becomes unreachable, but 4 is unaffected
3. and 4. are terminal, and can have no effect on other nodes

The code to generate the simple DAG:

```
import networkx as nx

G = nx.DiGraph()

# add 5 nodes, labeled 0-4:
```



```
map(G.add_node, range(5))
# 1,2 depend on 0:
G.add_edge(0,1)
G.add_edge(0,2)
# 3 depends on 1,2
G.add_edge(1,3)
G.add_edge(2,3)
# 4 depends on 1
G.add_edge(1,4)

# now draw the graph:
pos = { 0 : (0,0), 1 : (1,1), 2 : (-1,1),
        3 : (0,2), 4 : (2,2) }
nx.draw(G, pos, edge_color='r')
```

For demonstration purposes, we have a function that generates a random DAG with a given number of nodes and edges.

```
def random_dag(nodes, edges):
    """Generate a random Directed Acyclic Graph (DAG) with a given number of nodes and edges"""
    G = nx.DiGraph()
    for i in range(nodes):
        G.add_node(i)
    while edges > 0:
        a = randint(0,nodes-1)
        b=a
        while b==a:
            b = randint(0,nodes-1)
        G.add_edge(a,b)
        if nx.is_directed_acyclic_graph(G):
            edges -= 1
        else:
            # we closed a loop!
            G.remove_edge(a,b)
    return G
```

So first, we start with a graph of 32 nodes, with 128 edges:

```
In [2]: G = random_dag(32,128)
```

Now, we need to build our dict of jobs corresponding to the nodes on the graph:

```
In [3]: jobs = {}

# in reality, each job would presumably be different
# randomwait is just a function that sleeps for a random interval
In [4]: for node in G:
...:     jobs[node] = randomwait
```

Once we have a dict of jobs matching the nodes on the graph, we can start submitting jobs, and linking up the dependencies. Since we don't know a job's msg\_id until it is submitted, which is necessary for building dependencies, it is critical that we don't submit any jobs before other jobs it may depend on. Fortunately, NetworkX provides a `topological_sort()` method which ensures exactly this. It presents an iterable, that guarantees that when you arrive at a node, you have already visited all the nodes it on which it depends:

```

In [5]: rc = Client()
In [5]: view = rc.load_balanced_view()

In [6]: results = {}

In [7]: for node in G.topological_sort():
...:     # get list of AsyncResult objects from nodes
...:     # leading into this one as dependencies
...:     deps = [ results[n] for n in G.predecessors(node) ]
...:     # submit and store AsyncResult object
...:     results[node] = view.apply_with_flags(jobs[node], after=deps, block=False)

```

Now that we have submitted all the jobs, we can wait for the results:

```
In [8]: view.wait(results.values())
```

Now, at least we know that all the jobs ran and did not fail (`r.get()` would have raised an error if a task failed). But we don't know that the ordering was properly respected. For this, we can use the `metadata` attribute of each `AsyncResult`.

These objects store a variety of metadata about each task, including various timestamps. We can validate that the dependencies were respected by checking that each task was started after all of its predecessors were completed:

```

def validate_tree(G, results):
    """Validate that jobs executed after their dependencies."""
    for node in G:
        started = results[node].metadata.started
        for parent in G.predecessors(node):
            finished = results[parent].metadata.completed
            assert started > finished, "%s should have happened after %s"%(node, parent)

```

We can also validate the graph visually. By drawing the graph with each node's x-position as its start time, all arrows must be pointing to the right if dependencies were respected. For spreading, the y-position will be the runtime of the task, so long tasks will be at the top, and quick, small tasks will be at the bottom.

```

In [10]: from matplotlib.dates import date2num

In [11]: from matplotlib.cm import gist_rainbow

In [12]: pos = {}; colors = {}

In [12]: for node in G:
...:     md = results[node].metadata
...:     start = date2num(md.started)
...:     runtime = date2num(md.completed) - start
...:     pos[node] = (start, runtime)
...:     colors[node] = md.engine_id

In [13]: nx.draw(G, pos, node_list=colors.keys(), node_color=colors.values(),
...:             cmap=gist_rainbow)

```

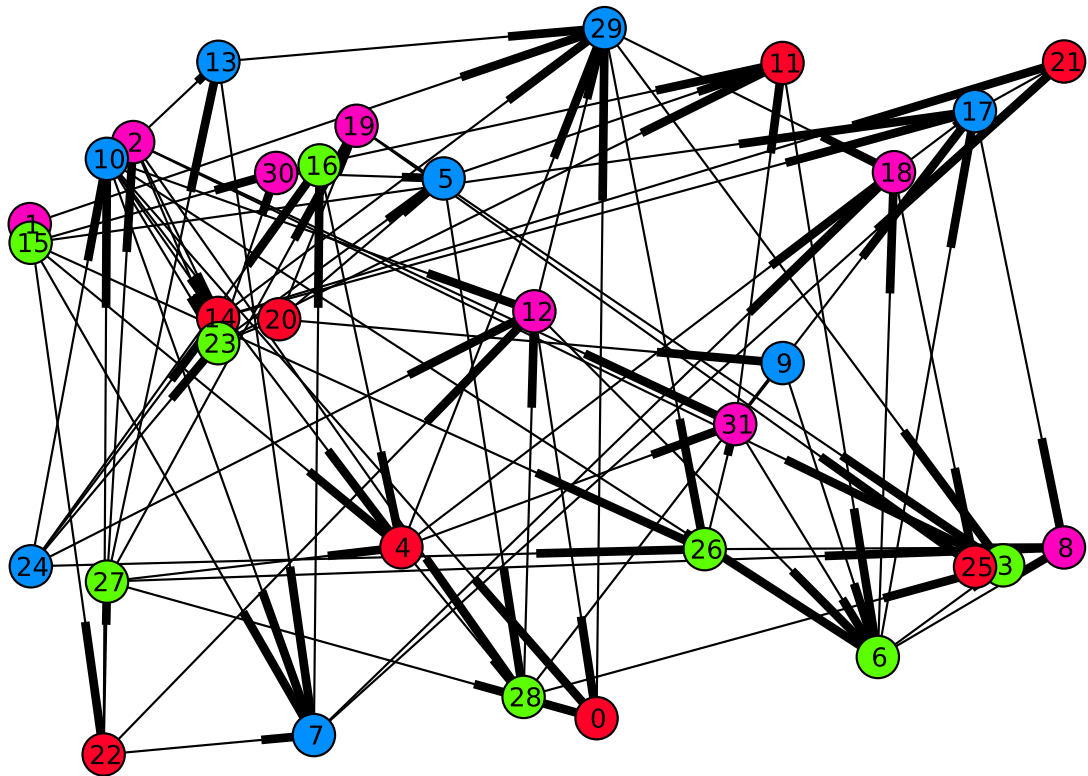


Figure 5.1: Time started on x, runtime on y, and color-coded by engine-id (in this case there were four engines). Edges denote dependencies.



## 5.10 Details of Parallel Computing with IPython

**Note:** There are still many sections to fill out

### 5.10.1 Caveats

First, some caveats about the detailed workings of parallel computing with OMQ and IPython.

#### Non-copying sends and numpy arrays

When numpy arrays are passed as arguments to apply or via data-movement methods, they are not copied. This means that you must be careful if you are sending an array that you intend to work on. PyZMQ does allow you to track when a message has been sent so you can know when it is safe to edit the buffer, but IPython only allows for this.

It is also important to note that the non-copying receive of a message is *read-only*. That means that if you intend to work in-place on an array that you have sent or received, you must copy it. This is true for both numpy arrays sent to engines and numpy arrays retrieved as results.

The following will fail:

```
In [3]: A = numpy.zeros(2)
```

```
In [4]: def setter(a):
...:     a[0]=1
...:     return a
```

```
In [5]: rc[0].apply_sync(setter, A)
```

```
-----
RemoteError                                Traceback (most recent call last)
```

```
...
```

```
RemoteError: RuntimeError(array is not writeable)
```

```
Traceback (most recent call last):
```

```
File "/path/to/site-packages/IPython/parallel/streamkernel.py", line 329, in apply_request
    exec code in working, working
```

```
File "<string>", line 1, in <module>
```

```
File "<ipython-input-14-736187483856>", line 2, in setter
```

```
RuntimeError: array is not writeable
```

If you do need to edit the array in-place, just remember to copy the array if it's read-only. The `ndarray.flags.writeable` flag will tell you if you can write to an array.

```
In [3]: A = numpy.zeros(2)
```

```
In [4]: def setter(a):
...:     """only copy read-only arrays"""
...:     if not a.flags.writeable:
...:         a=a.copy()
...:     a[0]=1
```

```
...:         return a

In [5]: rc[0].apply_sync(setter, A)
Out[5]: array([ 1.,  0.])

# note that results will also be read-only:
In [6]: _.flags.writeable
Out[6]: False
```

If you want to safely edit an array in-place after *sending* it, you must use the *track=True* flag. IPython always performs non-copying sends of arrays, which return immediately. You must instruct IPython track those messages *at send time* in order to know for sure that the send has completed. AsyncResults have a *sent* property, and *wait\_on\_send()* method for checking and waiting for OMQ to finish with a buffer.

```
In [5]: A = numpy.random.random((1024,1024))

In [6]: view.track=True

In [7]: ar = view.apply_async(lambda x: 2*x, A)

In [8]: ar.sent
Out[8]: False

In [9]: ar.wait_on_send() # blocks until sent is True
```

## What is sendable?

If IPython doesn't know what to do with an object, it will pickle it. There is a short list of objects that are not pickled: buffers, str/bytes objects, and numpy arrays. These are handled specially by IPython in order to prevent the copying of data. Sending bytes or numpy arrays will result in exactly zero in-memory copies of your data (unless the data is very small).

If you have an object that provides a Python buffer interface, then you can always send that buffer without copying - and reconstruct the object on the other side in your own code. It is possible that the object reconstruction will become extensible, so you can add your own non-copying types, but this does not yet exist.

## Closures

Just about anything in Python is pickleable. The one notable exception is objects (generally functions) with *closures*. Closures can be a complicated topic, but the basic principal is that functions that refer to variables in their parent scope have closures.

An example of a function that uses a closure:

```
def f(a):
    def inner():
        # inner will have a closure
        return a
    return inner()
```

```
f1 = f(1)
f2 = f(2)
f1() # returns 1
f2() # returns 2
```

`f1` and `f2` will have closures referring to the scope in which *inner* was defined, because they use the variable ‘`a`’. As a result, you would not be able to send `f1` or `f2` with IPython. Note that you *would* be able to send *f*. This is only true for interactively defined functions (as are often used in decorators), and only when there are variables used inside the inner function, that are defined in the outer function. If the names are *not* in the outer function, then there will not be a closure, and the generated function will look in `globals()` for the name:

```
def g(b):
    # note that 'b' is not referenced in inner's scope
    def inner():
        # this inner will not have a closure
        return a
    return echo
g1 = g(1)
g2 = g(2)
g1() # raises NameError on 'a'
a=5
g2() # returns 5
```

`g1` and `g2` will be sendable with IPython, and will treat the engine’s namespace as `globals()`. The `pull()` method is implemented based on this principal. If we did not provide `pull`, you could implement it yourself with *apply*, by simply returning objects out of the global namespace:

```
In [10]: view.apply(lambda : a)

# is equivalent to
In [11]: view.pull('a')
```

## 5.10.2 Running Code

There are two principal units of execution in Python: strings of Python code (e.g. ‘`a=5`’), and Python functions. IPython is designed around the use of functions via the core Client method, called *apply*.

### Apply

The principal method of remote execution is `apply()`, of View objects. The Client provides the full execution and communication API for engines via its low-level `send_apply_message()` method.

**f** [function] The function to be called remotely

**args** [tuple/list] The positional arguments passed to *f*

**kwargs** [dict] The keyword arguments passed to *f*

flags for all views:

**block** [bool (default: view.block)] Whether to wait for the result, or return immediately. False:

returns AsyncResult

**True:** returns actual result(s) of `f(*args, **kwargs)` if multiple targets:

list of results, matching *targets*

**track** [bool [default view.track]] whether to track non-copying sends.

**targets** [int,list of ints, 'all', None [default view.targets]] Specify the destination of the job. if 'all' or None:

Run on all active engines

**if list:** Run on each specified engine

**if int:** Run on single engine

Note that LoadBalancedView uses targets to restrict possible destinations. LoadBalanced calls will always execute in just one location.

flags only in LoadBalancedViews:

**after** [Dependency or collection of msg\_ids] Only for load-balanced execution (targets=None) Specify a list of msg\_ids as a time-based dependency. This job will only be run *after* the dependencies have been met.

**follow** [Dependency or collection of msg\_ids] Only for load-balanced execution (targets=None) Specify a list of msg\_ids as a location-based dependency. This job will only be run on an engine where this dependency is met.

**timeout** [float/int or None] Only for load-balanced execution (targets=None) Specify an amount of time (in seconds) for the scheduler to wait for dependencies to be met before failing with a DependencyTimeout.

## execute and run

For executing strings of Python code, `DirectView`'s also provide an `:meth:`execute` and a `run()` method, which rather than take functions and arguments, take simple strings. *execute* simply takes a string of Python code to execute, and sends it to the Engine(s). *run* is the same as *execute*, but for a *file*, rather than a string. It is simply a wrapper that does something very similar to `execute(open(f).read())`.

---

**Note:** TODO: Example

---

### 5.10.3 Views

The principal extension of the `Client` is the `View` class. The client

## DirectView

The `DirectView` is the class for the IPython *Multiplexing Interface*.

### Creating a DirectView

DirectViews can be created in two ways, by index access to a client, or by a client's `view()` method. Index access to a Client works in a few ways. First, you can create DirectViews to single engines simply by accessing the client by engine id:

```
In [2]: rc[0]
Out[2]: <DirectView 0>
```

You can also create a DirectView with a list of engines:

```
In [2]: rc[0,1,2]
Out[2]: <DirectView [0,1,2]>
```

Other methods for accessing elements, such as slicing and negative indexing, work by passing the index directly to the client's `ids` list, so:

```
# negative index
In [2]: rc[-1]
Out[2]: <DirectView 3>

# or slicing:
In [3]: rc[:,2]
Out[3]: <DirectView [0,2]>
```

are always the same as:

```
In [2]: rc[rc.ids[-1]]
Out[2]: <DirectView 3>

In [3]: rc[rc.ids[:,2]]
Out[3]: <DirectView [0,2]>
```

Also note that the slice is evaluated at the time of construction of the DirectView, so the targets will not change over time if engines are added/removed from the cluster.

### Execution via DirectView

The DirectView is the simplest way to work with one or more engines directly (hence the name).

### Data movement via DirectView

Since a Python namespace is just a dict, DirectView objects provide dictionary-style access by key and methods such as `get()` and `update()` for convenience. This make the remote namespaces of the engines appear as a local dictionary. Underneath, these methods call `apply()`:

```
In [51]: dview['a']=['foo','bar']
```

```
In [52]: dview['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 known as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `Client` 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]: dview.scatter('a', range(16))
```

```
Out[58]: [None, None, None, None]
```

```
In [59]: dview['a']
```

```
Out[59]: [ [0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15] ]
```

```
In [60]: dview.gather('a')
```

```
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

## Push and pull

push

pull

## LoadBalancedView

The `LoadBalancedView`

### 5.10.4 Data Movement

Reference

### 5.10.5 Results

#### AsyncResult

Our primary representation is the `AsyncResult` object, based on the object of the same name in the built-in `multiprocessing.pool` module. Our version provides a superset of that interface.

The basic principle of the `AsyncResult` is the encapsulation of one or more results not yet completed. Execution methods (including data movement, such as push/pull) will all return `AsyncResults` when `block=False`.

## The `mp.pool.AsyncResult` interface

The basic interface of the `AsyncResult` is exactly that of the `AsyncResult` in `multiprocessing.pool`, and consists of four methods:

### class `AsyncResult`

The `stdlib` `AsyncResult` spec

**`wait([timeout])`**

Wait until the result is available or until *timeout* seconds pass. This method always returns `None`.

**`ready()`**

Return whether the call has completed.

**`successful()`**

Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.

**`get([timeout])`**

Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised as a `RemoteError` by `get()`.

While an `AsyncResult` is not done, you can check on it with its `ready()` method, which will return whether the AR is done. You can also wait on an `AsyncResult` with its `wait()` method. This method blocks until the result arrives. If you don't want to wait forever, you can pass a timeout (in seconds) as an argument to `wait()`. `wait()` will *always return None*, and should never raise an error.

`ready()` and `wait()` are insensitive to the success or failure of the call. After a result is done, `successful()` will tell you whether the call completed without raising an exception.

If you actually want the result of the call, you can use `get()`. Initially, `get()` behaves just like `wait()`, in that it will block until the result is ready, or until a timeout is met. However, unlike `wait()`, `get()` will raise a `TimeoutError` if the timeout is reached and the result is still not ready. If the result arrives before the timeout is reached, then `get()` will return the result itself if no exception was raised, and will raise an exception if there was.

Here is where we start to expand on the `multiprocessing` interface. Rather than raising the original exception, a `RemoteError` will be raised, encapsulating the remote exception with some metadata. If the `AsyncResult` represents multiple calls (e.g. any time *targets* is plural), then a `CompositeError`, a subclass of `RemoteError`, will be raised.

### See Also:

For more information on remote exceptions, see *the section in the Direct Interface*.

## Extended interface

Other extensions of the `AsyncResult` interface include convenience wrappers for `get()`. `AsyncResult`s have a property, `result`, with the short alias `r`, which simply call `get()`. Since our object is designed for representing *parallel* results, it is expected that many calls (any of those submitted via `DirectView`) will

map results to engine IDs. We provide a `get_dict()`, which is also a wrapper on `get()`, which returns a dictionary of the individual results, keyed by engine ID.

You can also prevent a submitted job from actually executing, via the `AsyncResult`'s `abort()` method. This will instruct engines to not execute the job when it arrives.

The larger extension of the `AsyncResult` API is the `metadata` attribute. The metadata is a dictionary (with attribute access) that contains, logically enough, metadata about the execution.

Metadata keys:

`timestamps`

**submitted** When the task left the Client

**started** When the task started execution on the engine

**completed** When execution finished on the engine

**received** When the result arrived on the Client

note that it is not known when the result arrived in OMQ on the client, only when it arrived in Python via `Client.spin()`, so in interactive use, this may not be strictly informative.

Information about the engine

**engine\_id** The integer id

**engine\_uuid** The UUID of the engine

output of the call

**pyerr** Python exception, if there was one

**pyout** Python output

**stderr** stderr stream

**stdout** stdout (e.g. `print`) stream

And some extended information

**status** either 'ok' or 'error'

**msg\_id** The UUID of the message

**after** For tasks: the time-based `msg_id` dependencies

**follow** For tasks: the location-based `msg_id` dependencies

While in most cases, the Clients that submitted a request will be the ones using the results, other Clients can also request results directly from the Hub. This is done via the Client's `get_result()` method. This method will *always* return an `AsyncResult` object. If the call was not submitted by the client, then it will be a subclass, called `AsyncHubResult`. These behave in the same way as an `AsyncResult`, but if the result is not ready, waiting on an `AsyncHubResult` polls the Hub, which is much more expensive than the passive polling used in regular `AsyncResults`.

The Client keeps track of all results history, results, metadata



### 5.10.6 Querying the Hub

The Hub sees all traffic that may pass through the schedulers between engines and clients. It does this so that it can track state, allowing multiple clients to retrieve results of computations submitted by their peers, as well as persisting the state to a database.

`queue_status`

You can check the status of the queues of the engines with this command.

`result_status`

check on results

`purge_results`

forget results (conserve resources)

### 5.10.7 Controlling the Engines

There are a few actions you can do with Engines that do not involve execution. These messages are sent via the Control socket, and bypass any long queues of waiting execution jobs

`abort`

Sometimes you may want to prevent a job you have submitted from actually running. The method for this is `abort()`. It takes a container of `msg_ids`, and instructs the Engines to not run the jobs if they arrive. The jobs will then fail with an `AbortedTask` error.

`clear`

You may want to purge the Engine(s) namespace of any data you have left in it. After running *clear*, there will be no names in the Engine's namespace

`shutdown`

You can also instruct engines (and the Controller) to terminate from a Client. This can be useful when a job is finished, since you can shutdown all the processes with a single command.

### 5.10.8 Synchronization

Since the Client is a synchronous object, events do not automatically trigger in your interactive session - you must poll the ZMQ sockets for incoming messages. Note that this polling *does not* actually make any network requests. It simply performs a *select* operation, to check if messages are already in local memory, waiting to be handled.

The method that handles incoming messages is `spin()`. This method flushes any waiting messages on the various incoming sockets, and updates the state of the Client.

If you need to wait for particular results to finish, you can use the `wait()` method, which will call `spin()` until the messages are no longer outstanding. Anything that represents a collection of messages, such as a list of `msg_ids` or one or more `AsyncResult` objects, can be passed as argument to `wait`. A timeout can be

specified, which will prevent the call from blocking for more than a specified time, but the default behavior is to wait forever.

The client also has an *outstanding* attribute - a `set` of `msg_ids` that are awaiting replies. This is the default if `wait` is called with no arguments - i.e. wait on *all* outstanding messages.

---

**Note:** TODO wait example

---

### 5.10.9 Map

Many parallel computing problems can be expressed as a *map*, or running a single program with a variety of different inputs. Python has a built-in **:py-func:'map'**, which does exactly this, and many parallel execution tools in Python, such as the built-in **:py-class:'multiprocessing.Pool'** object provide implementations of *map*. All View objects provide a `map()` method as well, but the load-balanced and direct implementations differ.

Views' `map` methods can be called on any number of sequences, but they can also take the *block* and *bound* keyword arguments, just like `apply()`, but *only as keywords*.

```
dview.map(*sequences, block=None)
```

- `iter`, `map_async`, `reduce`

### 5.10.10 Decorators and RemoteFunctions

`@parallel`

`@remote`

`RemoteFunction`

`ParallelFunction`

### 5.10.11 Dependencies

`@depend`

`@require`

`Dependency`

## 5.11 Transitioning from IPython.kernel to IPython.parallel

We have rewritten our parallel computing tools to use `0MQ` and `Tornado`. The redesign has resulted in dramatically improved performance, as well as (we think), an improved interface for executing code remotely. This doc is to help users of `IPython.kernel` transition their codes to the new code.

### 5.11.1 Processes

The process model for the new parallel code is very similar to that of IPython.kernel. There is still a Controller, Engines, and Clients. However, the the Controller is now split into multiple processes, and can even be split across multiple machines. There does remain a single ipcontroller script for starting all of the controller processes.

---

**Note:** TODO: fill this out after config system is updated

---

#### See Also:

Detailed *Parallel Process* doc for configuring and launching IPython processes.

### 5.11.2 Creating a Client

Creating a client with default settings has not changed much, though the extended options have. One significant change is that there are no longer multiple Client classes to represent the various execution models. There is just one low-level Client object for connecting to the cluster, and View objects are created from that Client that provide the different interfaces for execution.

To create a new client, and set up the default direct and load-balanced objects:

```
# old
In [1]: from IPython.kernel import client as kclient

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

In [3]: tc = kclient.TaskClient()

# new
In [1]: from IPython.parallel import Client

In [2]: rc = Client()

In [3]: dview = rc[:]

In [4]: lbview = rc.load_balanced_view()
```

### 5.11.3 Apply

The main change to the API is the addition of the `apply()` to the View objects. This is a method that takes `view.apply(f,*args,**kwargs)`, and calls `f(*args, **kwargs)` remotely on one or more engines, returning the result. This means that the natural unit of remote execution is no longer a string of Python code, but rather a Python function.

- non-copying sends (track)
- remote References

The flags for execution have also changed. Previously, there was only *block* denoting whether to wait for results. This remains, but due to the addition of fully non-copying sends of arrays and buffers, there is also a *track* flag, which instructs PyZMQ to produce a `MessageTracker` that will let you know when it is safe again to edit arrays in-place.

The result of a non-blocking call to *apply* is now an **AsyncResult\_** object, described below.

#### 5.11.4 MultiEngine to DirectView

The multiplexing interface previously provided by the `MultiEngineClient` is now provided by the `DirectView`. Once you have a `Client` connected, you can create a `DirectView` with index-access to the client (`view = client[1:5]`). The core methods for communicating with engines remain: *execute*, *run*, *push*, *pull*, *scatter*, *gather*. These methods all behave in much the same way as they did on a `MultiEngineClient`.

```
# old
In [2]: mec.execute('a=5', targets=[0,1,2])

# new
In [2]: view.execute('a=5', targets=[0,1,2])
# or
In [2]: rc[0,1,2].execute('a=5')
```

This extends to any method that communicates with the engines.

Requests of the Hub (queue status, etc.) are no-longer asynchronous, and do not take a *block* argument.

- `get_ids()` is now the property `ids`, which is passively updated by the Hub (no need for network requests for an up-to-date list).
- `barrier()` has been renamed to `wait()`, and now takes an optional timeout. `flush()` is removed, as it is redundant with `wait()`
- `zip_pull()` has been removed
- `keys()` has been removed, but is easily implemented as:

```
dview.apply(lambda : globals().keys())
```

- `push_function()` and `push_serialized()` are removed, as `push()` handles functions without issue.

#### See Also:

*Our [Direct Interface doc](#) for a simple tutorial with the `DirectView`.*

The other major difference is the use of `apply()`. When remote work is simply functions, the natural return value is the actual Python objects. It is no longer the recommended pattern to use `stdout` as your results, due to stream decoupling and the asynchronous nature of how the `stdout` streams are handled in the new system.

### 5.11.5 Task to LoadBalancedView

Load-Balancing has changed more than Multiplexing. This is because there is no longer a notion of a `StringTask` or a `MapTask`, there are simply Python functions to call. Tasks are now simpler, because they are no longer composites of push/execute/pull/clear calls, they are a single function that takes arguments, and returns objects.

The load-balanced interface is provided by the `LoadBalancedView` class, created by the client:

```
In [10]: lbview = rc.load_balanced_view()

# load-balancing can also be restricted to a subset of engines:
In [10]: lbview = rc.load_balanced_view([1,2,3])
```

A simple task would consist of sending some data, calling a function on that data, plus some data that was resident on the engine already, and then pulling back some results. This can all be done with a single function.

Let's say you want to compute the dot product of two matrices, one of which resides on the engine, and another resides on the client. You might construct a task that looks like this:

```
In [10]: st = kclient.StringTask("""
import numpy
C=numpy.dot(A,B)
""",
push=dict(B=B),
pull='C'
)
```

```
In [11]: tid = tc.run(st)
```

```
In [12]: tr = tc.get_task_result(tid)
```

```
In [13]: C = tc['C']
```

In the new code, this is simpler:

```
In [10]: import numpy
```

```
In [11]: from IPython.parallel import Reference
```

```
In [12]: ar = lbview.apply(numpy.dot, Reference('A'), B)
```

```
In [13]: C = ar.get()
```

Note the use of `Reference`. This is a convenient representation of an object that exists in the engine's namespace, so you can pass remote objects as arguments to your task functions.

Also note that in the kernel model, after the task is run, 'A', 'B', and 'C' are all defined on the engine. In order to deal with this, there is also a `clear_after` flag for Tasks to prevent pollution of the namespace, and bloating of engine memory. This is not necessary with the new code, because only those objects explicitly pushed (or set via `globals()`) will be resident on the engine beyond the duration of the task.

See Also:

Dependencies also work very differently than in IPython.kernel. See our *[doc on Dependencies](#)* for details.

**See Also:**

*[Our Task Interface doc](#)* for a simple tutorial with the LoadBalancedView.

**PendingResults to AsyncResults**

With the departure from Twisted, we no longer have the `Deferred` class for representing unfinished results. For this, we have an `AsyncResult` object, based on the object of the same name in the built-in `multiprocessing.pool` module. Our version provides a superset of that interface.

However, unlike in IPython.kernel, we do not have `PendingDeferred`, `PendingResult`, or `TaskResult` objects. Simply this one object, the `AsyncResult`. Every asynchronous (*block=False*) call returns one.

The basic methods of an `AsyncResult` are:

```
AsyncResult.wait([timeout]): # wait for the result to arrive
AsyncResult.get([timeout]): # wait for the result to arrive, and then return it
AsyncResult.metadata: # dict of extra information about execution.
```

There are still some things that behave the same as IPython.kernel:

```
# old
In [5]: pr = mec.pull('a', targets=[0,1], block=False)
In [6]: pr.r
Out[6]: [5, 5]

# new
In [5]: ar = dview.pull('a', targets=[0,1], block=False)
In [6]: ar.r
Out[6]: [5, 5]
```

The `.r` or `.result` property simply calls `get()`, waiting for and returning the result.

**See Also:**

*[AsyncResult details](#)*

---

# 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 a platform-specific default.

On posix systems (Linux, Unix, etc.), IPython respects the `$XDG_CONFIG_HOME` part of the [XDG Base Directory](#) specification. If `$XDG_CONFIG_HOME` is defined and exists ( `XDG_CONFIG_HOME` has a default interpretation of `$HOME/.config`), then IPython's config directory will be located in `$XDG_CONFIG_HOME/ipython`. If users still have an IPython directory in `$HOME/.ipython`, then that will be used. in preference to the system default.

For most users, the default value will simply be something like `$HOME/.config/ipython` on Linux, or `$HOME/.ipython` elsewhere.

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 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.

### 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 = [
    ('!a', '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 `$HOME/.config/ipython` if `$HOME/.config` exists (Linux), or `$HOME/.ipython` as a secondary default. 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 \$IPYTHON\_DIR/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 paggers (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` `'${commands.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 <a href="#">msysGit</a>
OS X	Use the <a href="#">git-osx-installer</a>

#### 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](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.

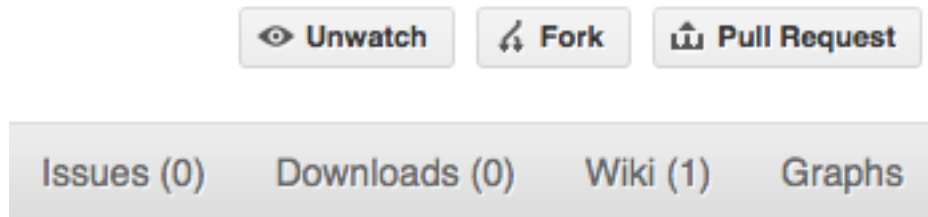
### Set up and configure a github account

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:

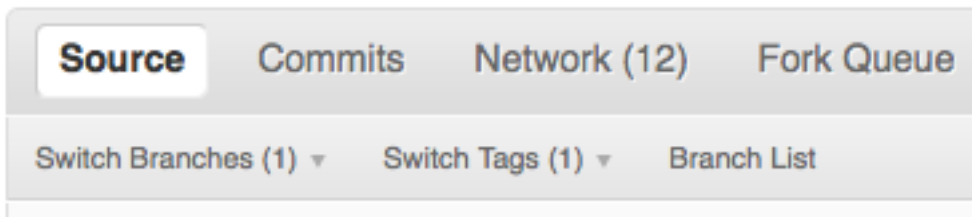
```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be 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.

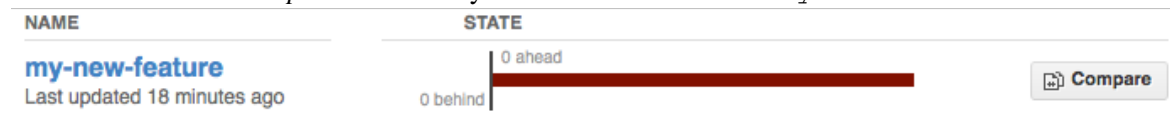
6. To push the changes up to your forked repo on [github](#), do a `git push` (see *git push*).

### 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`:



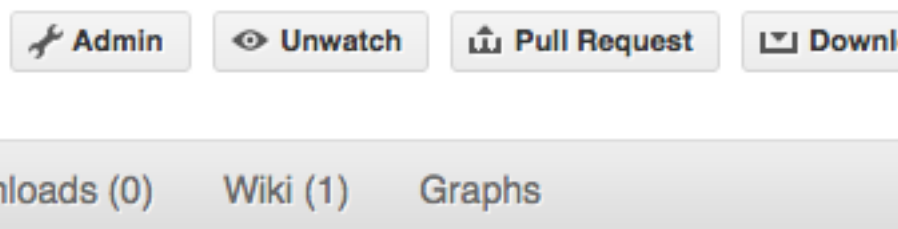
4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the [ipython mailing list](#), 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
```

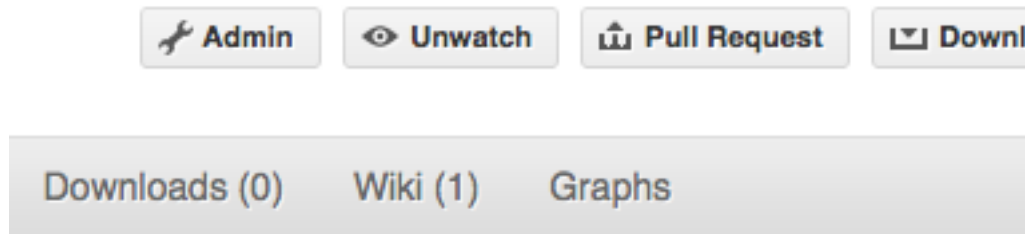
(Note the colon `:` before `test-branch`. See also: <http://github.com/guides/remove-a-remote-branch>)

## 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/.config/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.

---

### 7.4.3 Building and uploading

The built docs are stored in a separate repository. Through some github magic, they're automatically exposed as a website. It works like this:

- You will need to have sphinx and latex installed. In Ubuntu, install `texlive-latex-recommended texlive-latex-extra`



`texlive-fonts-recommended`. Install the latest version of sphinx from PyPI (`pip install sphinx`).

- Ensure that the development version of IPython is the first in your system path. You can either use a `virtualenv`, or modify your `PYTHONPATH`.
- Switch into the docs directory, and run `make gh-pages`. This will build your updated docs as html and pdf, then automatically check out the latest version of the docs repository, copy the built docs into it, and commit your changes.
- Open the built docs in a web browser, and check that they're as expected.
- (When building the docs for a new tagged release, you will have to add its link to `index.rst`, then run `python build_index.py` to update `index.html`. Commit the change.)
- Upload the docs with `git push`. This only works if you have write access to the docs repository.
- If you are building a version that is not the current dev branch, nor a tagged release, then you must run `gh-pages.py` directly with `python gh-pages.py <version>`, and *not* with `make gh-pages`.

## 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(unittest.TestCase):

    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

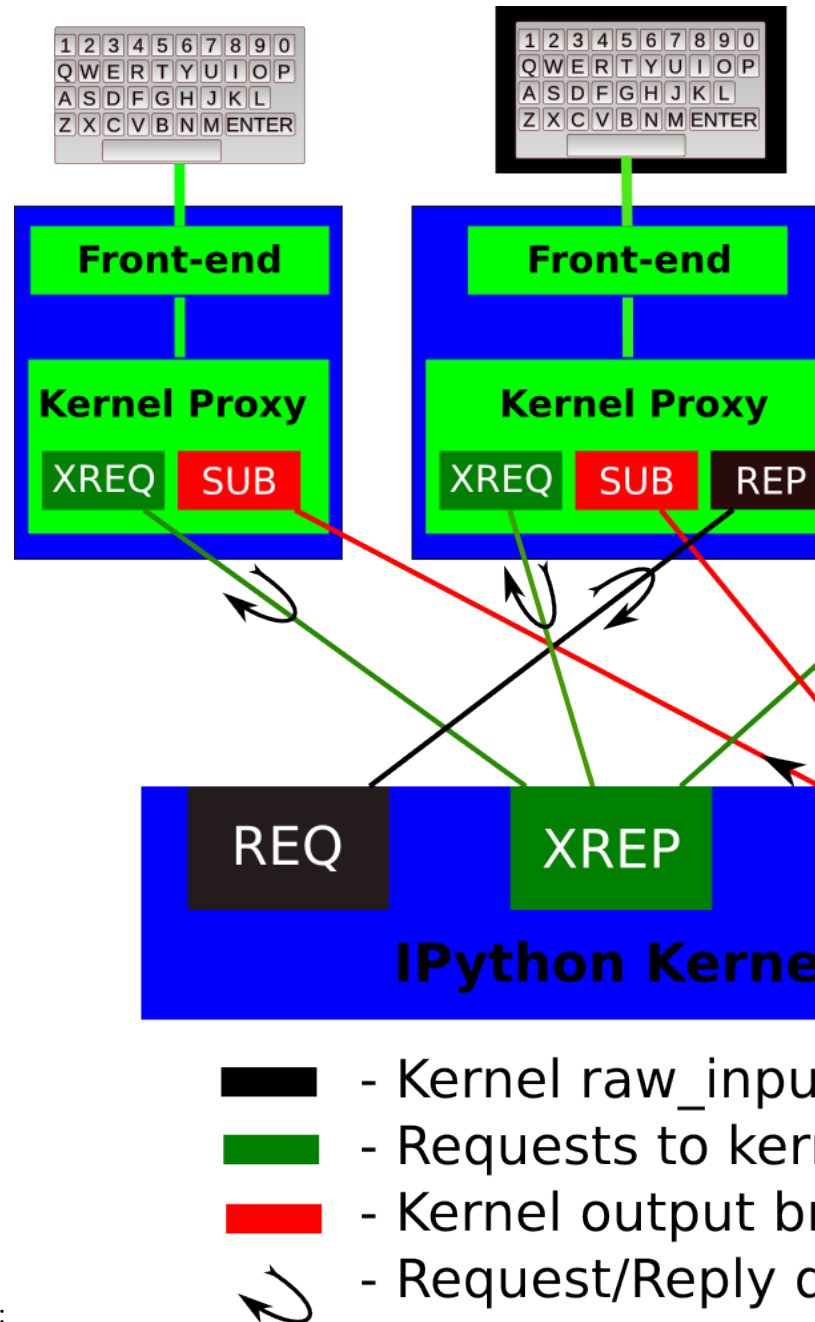
### 7.9.1 Introduction

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 collection of processes in the IPython parallel computing model. It has two major components:

- The Hub
- A collection of Schedulers

### 7.10.2 The Hub

The Hub is the central process for monitoring the state of the engines, and all task requests and results. It has no role in execution and does no relay of messages, so large blocking requests or database actions in the Hub do not have the ability to impede job submission and results.

#### Registration (XREP)

The first function of the Hub 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 MUX queue zmq.IDENTITY
    'control'    : 'abcd-1234-...', # the control queue zmq.IDENTITY
    'heartbeat'  : 'abcd-1234-...' # the heartbeat zmq.IDENTITY
}
```

---

**Note:** these are always the same, at least for now.

---

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 queue 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
}
```

```
'control' : 'tcp://...', # addr for control queue
'heartbeat' : ('tcp://...', 'tcp://...'), # tuple containing two interfaces needed for i
'task' : 'tcp://...', # addr for task queue, or None if no task queue running
}
```

Clients use the same socket as engines 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 hub 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 MUX queue
    'task' : ('lru', 'tcp...'), # routing scheme and addr for task queue (len 2 tuple)
    'query' : 'tcp...', # addr for methods to query the hub, like queue_request, etc.
    'control' : 'tcp...', # addr for control methods, like abort, etc.
}
```

## Heartbeat

The hub uses a heartbeat system to monitor engines, and track when they become unresponsive. As described in *messaging*, and shown in *connections*.

## Notification (PUB)

The hub publishes 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 and Control Queues.

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 hub monitors and logs all queue traffic, so that clients can retrieve past results or monitor pending tasks. This information may reside in-memory on the Hub, or on disk in a database (SQLite and MongoDB are currently supported). These requests are handled by the same socket as registration.

`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 `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. The three keys of each dict are:

```
'completed' : messages submitted via any queue that ran on the engine
'queue' : jobs submitted via MUX queue, whose results have not been received
'tasks' : tasks that are known to have been submitted to the engine, but
          have not completed. Note that with the pure zmq scheduler, this will
          always be 0/[].
```

Message type: `queue_reply`:

```
content = {
    'status' : 'ok', # or 'error'
    # if verbose=False:
    '0' : {'completed' : 1, 'queue' : 7, 'tasks' : 0},
    # if verbose=True:
    '1' : {'completed' : ['abcd-...', '1234-...'], 'queue' : ['58008-'], 'tasks' : []},
}
```

Clients can request individual results directly from the hub. This is primarily for gathering results of executions not submitted by the requesting 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*. An additional boolean key *'statusonly'* can be used to not request the results, but simply poll the status of the jobs.

Message type: `result_request`:

```
content = {
    'msg_ids' : ['uuid', '...'], # list of strs
    'targets' : [1,2,3], # list of int ids or uuids
    'statusonly' : False, # bool
}
```

The `result_request()` reply contains the content objects of the actual execution reply messages. If *statusonly=True*, then there will be only the *'pending'* and *'completed'* lists.

Message type: `result_reply`:

```

content = {
    'status' : 'ok', # else error
    # if ok:
    'acbd-...' : msg, # the content dict is keyed by msg_ids,
                    # values are the result messages
                    # there will be none of these if 'statusonly=True'
    'pending' : ['msg_id', '...'], # msg_ids still pending
    'completed' : ['msg_id', '...'], # list of completed msg_ids
}
buffers = ['bufs', '...'] # the buffers that contained the results of the objects.
                        # this will be empty if no messages are complete, or if
                        # statusonly is True.

```

For memory management purposes, Clients can also instruct the hub 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. This may no longer be necessary with the mongodb-based message logging backend.

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'
}

```

### 7.10.3 Schedulers

There are three basic schedulers:

- Task Scheduler
- MUX Scheduler
- Control Scheduler

The MUX and Control schedulers are simple MonitoredQueue ØMQ devices, with XREP sockets on either side. This allows the queue to relay individual messages to particular targets via `zmq.IDENTITY` routing. The Task scheduler may be a MonitoredQueue ØMQ device, in which case the client-facing socket is XREP, and the engine-facing socket is XREQ. The result of this is that client-submitted messages are load-balanced via the XREQ socket, but the engine's replies to each message go to the requesting client.

Raw XREQ scheduling is quite primitive, and doesn't allow message introspection, so there are also Python Schedulers that can be used. These Schedulers behave in much the same way as a MonitoredQueue does

from the outside, but have rich internal logic to determine destinations, as well as handle dependency graphs. Their sockets are always XREP on both sides.

The Python task schedulers have an additional message type, which informs the Hub of the destination of a task as soon as that destination is known.

Message type: `task_destination`:

```
content = {
    'msg_id' : 'abcd-1234-...', # the msg's uuid
    'engine_id' : '1234-abcd-...', # the destination engine's zmq.IDENTITY
}
```

### `apply()` and `apply_bound()`

In terms of message classes, the MUX scheduler and Task scheduler relay the exact same message types. Their only difference lies in how the destination is selected.

The `Namespace` model suggests that execution be able to use the model:

```
ns.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 a sendable buffer format for minimal data copying (exactly zero copies of numpy array data or buffers or large strings).

Message type: `apply_request`:

```
content = {
    'bound' : True, # whether to execute in the engine's namespace or unbound
    'after' : ['msg_id',...], # list of msg_ids or output of Dependency.as_dict()
    'follow' : ['msg_id',...], # list of msg_ids or output of Dependency.as_dict()
}

buffers = ['...'] # at least 3 in length
                # as built by build_apply_message(f,args,kwargs)
```

after/follow represent task dependencies. 'after' corresponds to a time dependency. The request will not arrive at an engine until the 'after' dependency tasks have completed. 'follow' corresponds to a location dependency. The task will be submitted to the same engine as these `msg_ids` (see `Dependency` docs for details).

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'
```

All engine execution and data movement is performed via apply messages.

#### 7.10.4 Control Messages

Messages that interact with the engines, but are not meant to execute code, are submitted via the Control queue. These messages have high priority, and are thus received and handled before any execution requests.

Clients may want to clear the namespace on the engine. There are no arguments nor information involved in this request, so the content is empty.

Message type: `clear_request`:

```
content = {}
```

Message type: `clear_reply`:

```
content = {  
    'status' : 'ok' # 'ok' or 'error'  
    # other error info here, as in other messages  
}
```

Clients may want to abort tasks that have not yet run. This can be done by message id, or all enqueued messages can be aborted if None is specified.

Message type: `abort_request`:

```
content = {  
    'msg_ids' : ['1234-...', '...'] # list of msg_ids or None  
}
```

Message type: `abort_reply`:

```
content = {  
    'status' : 'ok' # 'ok' or 'error'  
    # other error info here, as in other messages  
}
```

The last action a client may want to do is shutdown the kernel. If a kernel receives a shutdown request, then it aborts all queued messages, replies to the request, and exits.

Message type: `shutdown_request`:

```
content = {}
```

Message type: `shutdown_reply`:

```
content = {  
    'status' : 'ok' # 'ok' or 'error'
```

```
# other error info here, as in other messages
}
```

### 7.10.5 Implementation

There are a few differences in implementation between the *StreamSession* object used in the *newparallel* branch 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 hub inspects all messages, and doesn't need to see the content of the messages, which can be large, messages are now 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 IPython's *newparallel* branch on GitHub.

The IPython cluster consists of a Controller, and one or more each of clients and engines. The goal of the Controller is to manage and monitor the connections and communications between the clients and the engines. The Controller is no longer a single process entity, but rather a collection of processes - specifically one Hub, and 3 (or more) Schedulers.

It is important for security/practicality reasons that all connections be inbound to the controller processes. The arrows in the figures indicate the direction of the connection.

The Controller consists of 1-4 processes. Central to the cluster is the **Hub**, which monitors engine state, execution traffic, and handles registration and notification. The Hub includes a Heartbeat Monitor for keeping track of engines that are alive. Outside the Hub are 4 **Schedulers**. These devices are very small pure-C MonitoredQueue processes (or optionally threads) that relay messages very fast, but also send a copy of each message along a side socket to the Hub. The MUX queue and Control queue are MonitoredQueue ØMQ devices which relay explicitly addressed messages from clients to engines, and their replies back up. The Balanced queue performs load-balancing destination-agnostic scheduling. It may be a MonitoredQueue device, but may also be a Python Scheduler that behaves externally in an identical fashion to MQ devices, but

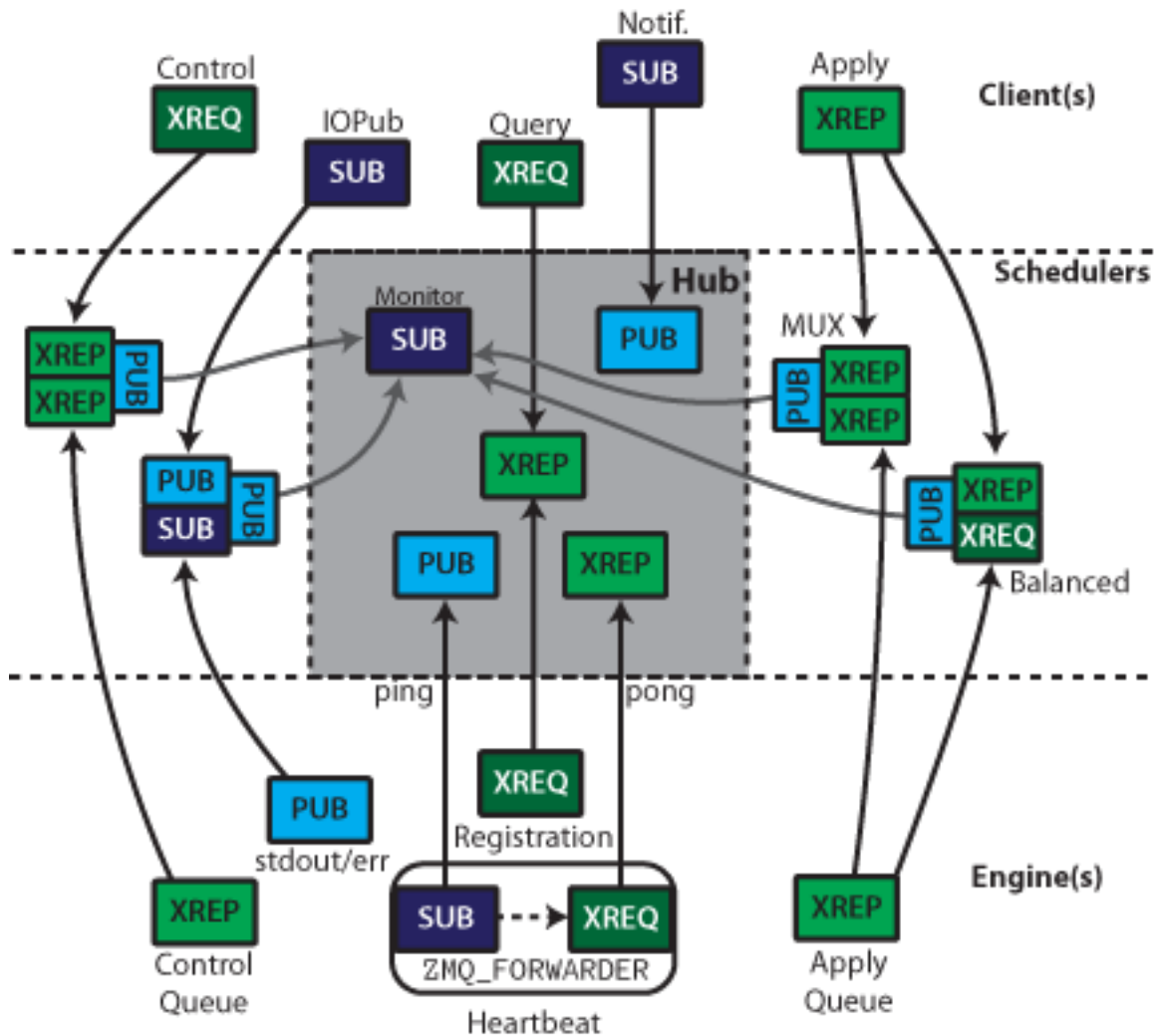


Figure 7.1: All the connections involved in connecting one client to one engine.

with additional internal logic. stdout/err are also propagated from the Engines to the clients via a PUB/SUB MonitoredQueue.

## Registration

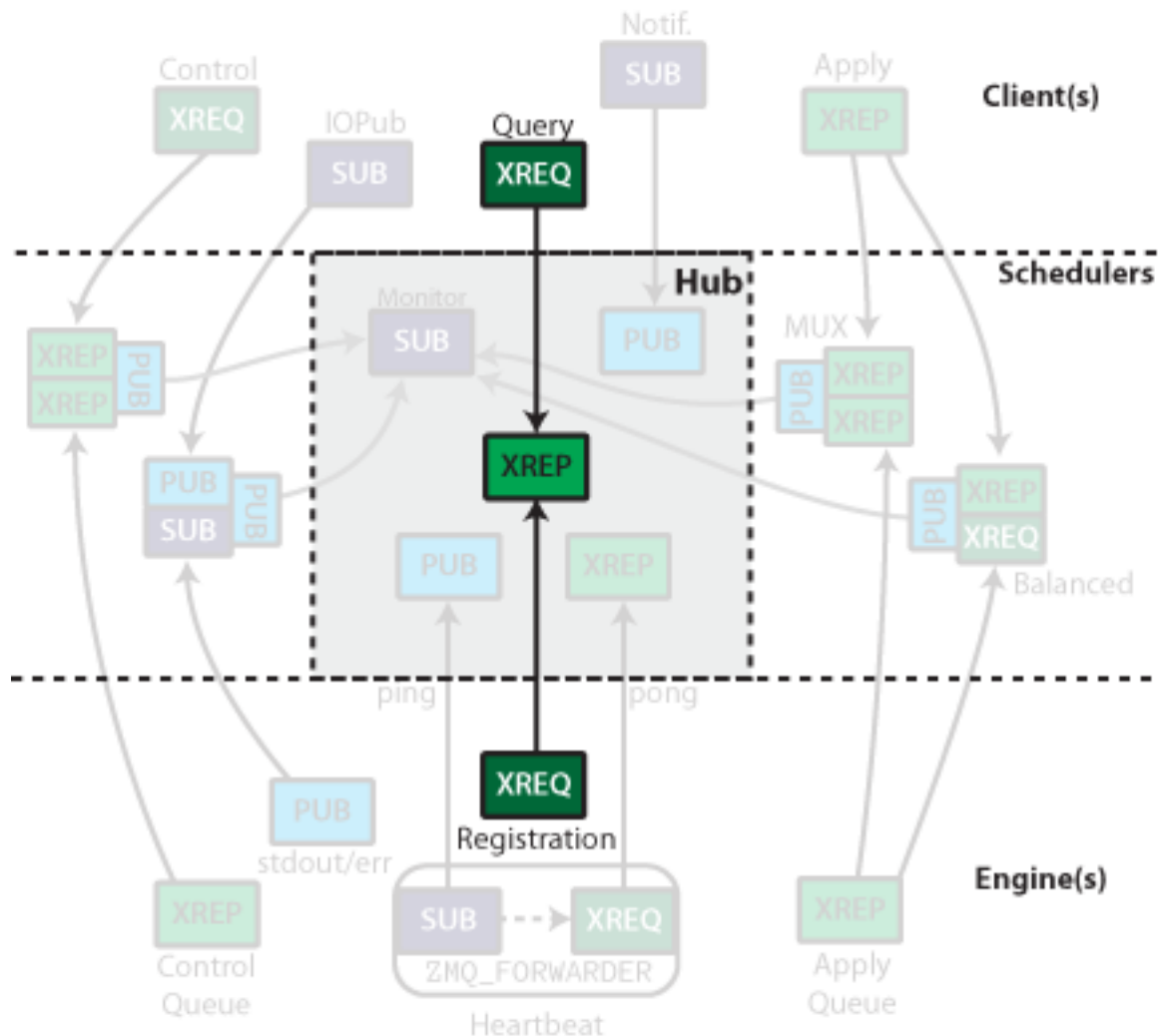


Figure 7.2: Engines and Clients only need to know where the Query XREP is located to start connecting.

Once a controller is launched, the only information needed for connecting clients and/or engines is the IP/port of the Hub's 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. Clients use this same socket for querying the Hub for state information.

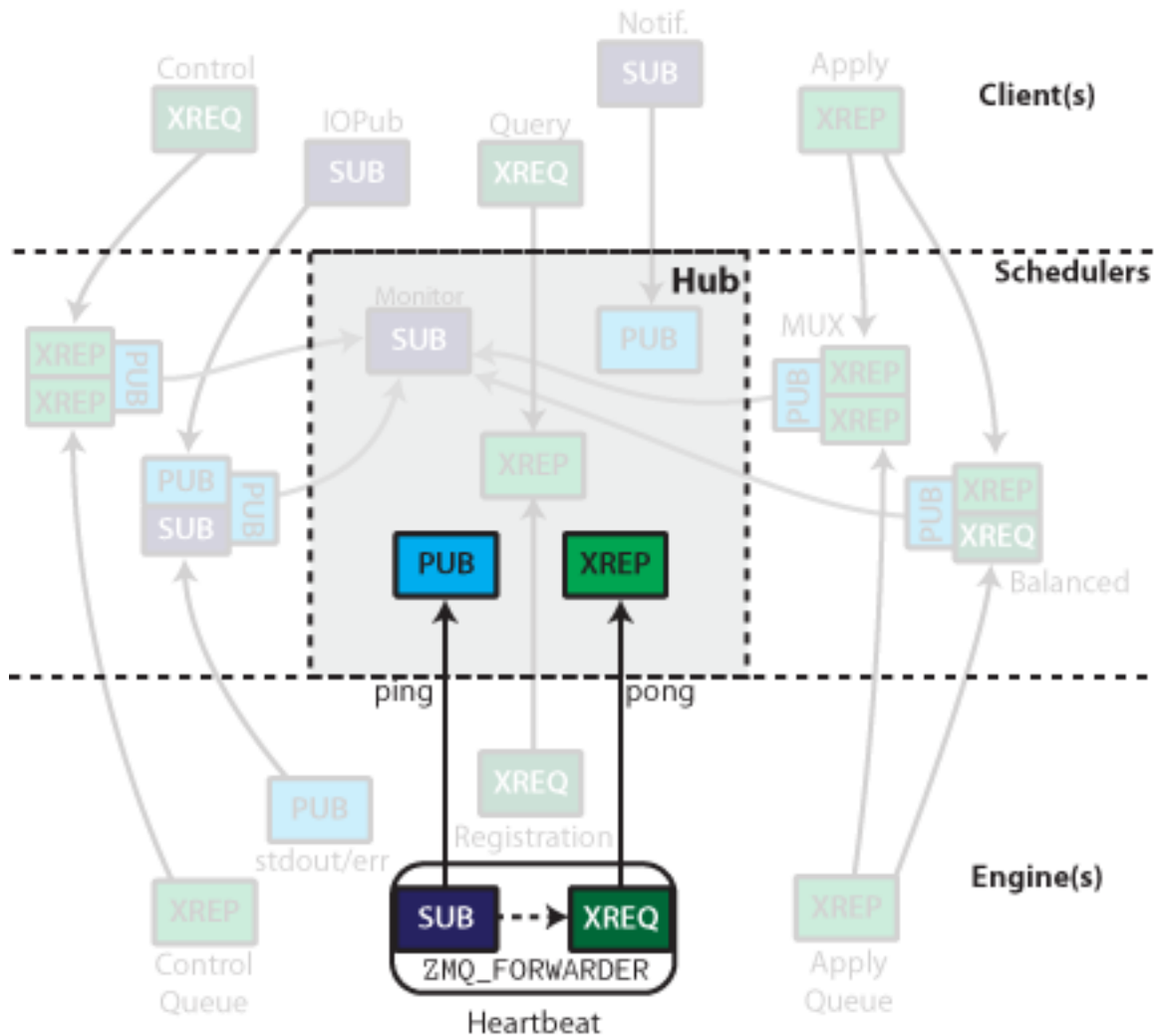


Figure 7.3: The heartbeat sockets.

## Heartbeat

The heartbeat process has been described elsewhere. To summarize: the Heartbeat Monitor 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 *ping*, and the XREQ is connected to the XREP labeled *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, which information the Hub uses to notify clients of any changes in the available engines.

## Schedulers

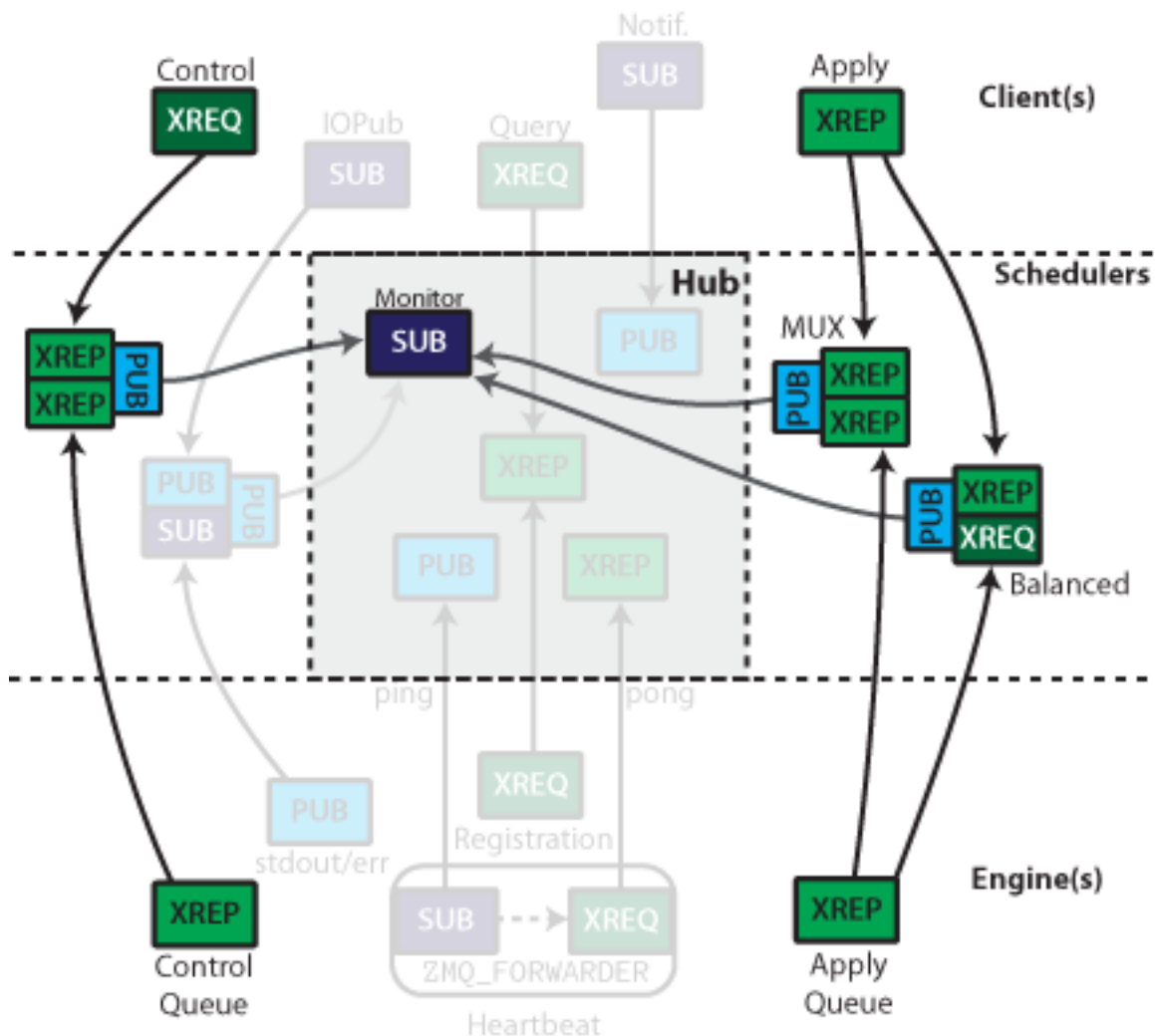


Figure 7.4: Control message scheduler on the left, execution (apply) schedulers on the right.

The controller has at least three Schedulers. These devices are primarily for relaying messages between clients and engines, but the Hub 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 Hub 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 XREP socket connected to the downstream end of the device.

At the Kernel level, both of these XREP sockets are treated in the same way as the REP socket in the serial version (except using ZMQStreams instead of explicit sockets).

Execution can be done in a load-balanced (engine-agnostic) or multiplexed (engine-specified) manner. The sockets on the Client and Engine are the same for these two actions, but the scheduler used determines the actual behavior. This routing is done via the `zmq.IDENTITY` of the upstream sockets in each MonitoredQueue.

## IOPub



Figure 7.5: stdout/err are published via a PUB/SUB MonitoredQueue

On the kernels, stdout/stderr are captured and published via a PUB socket. These PUB sockets all connect to a SUB socket input of a MonitoredQueue, which subscribes to all messages. They are then republished via another PUB socket, which can be subscribed by the clients.

## Client connections

The hub's registrar XREP socket also listens for queries from clients as to queue status, and control instructions. Clients connect to this socket via an XREQ during registration.

The Hub 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.

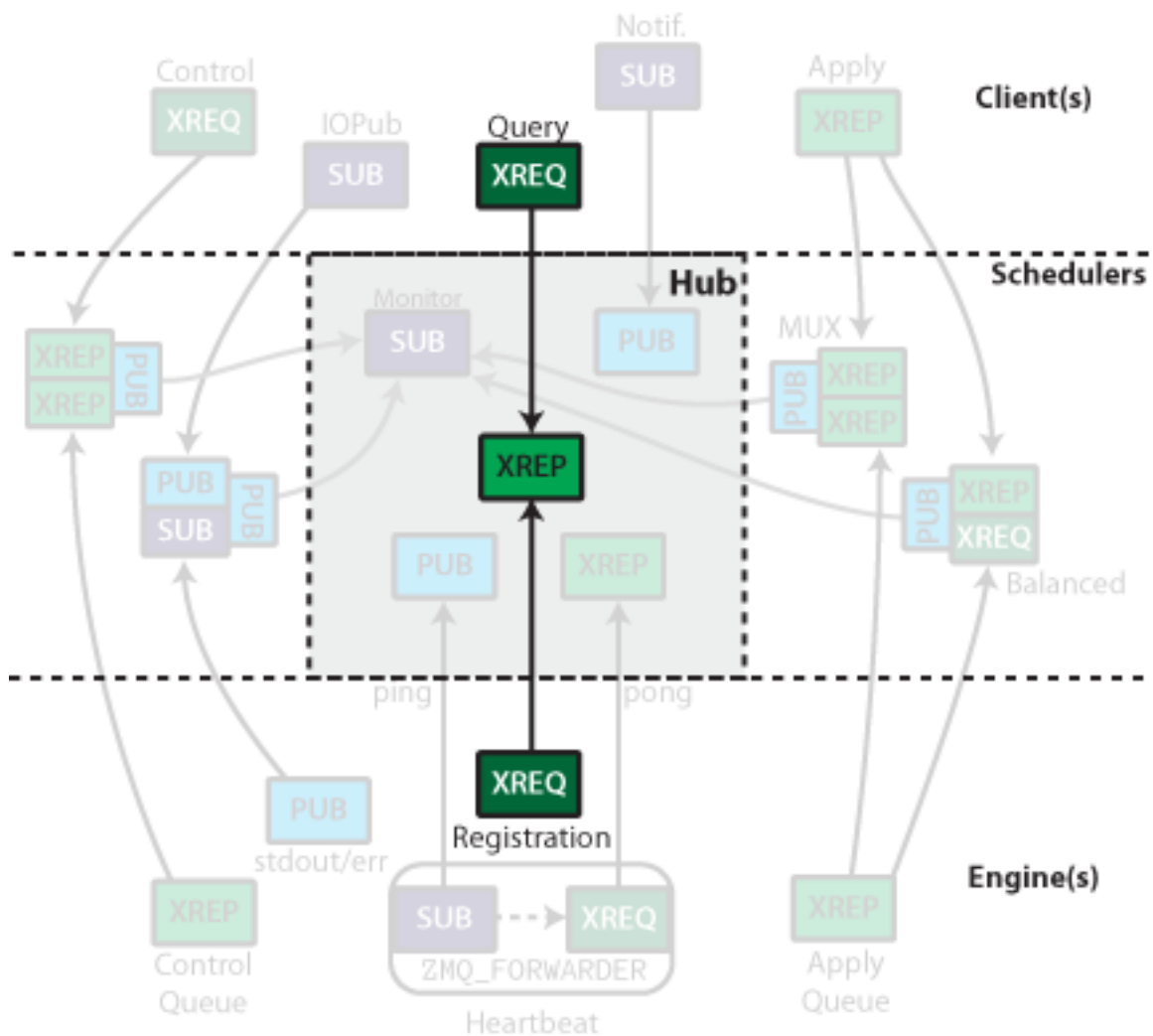


Figure 7.6: Clients connect to an XREP socket to query the hub.



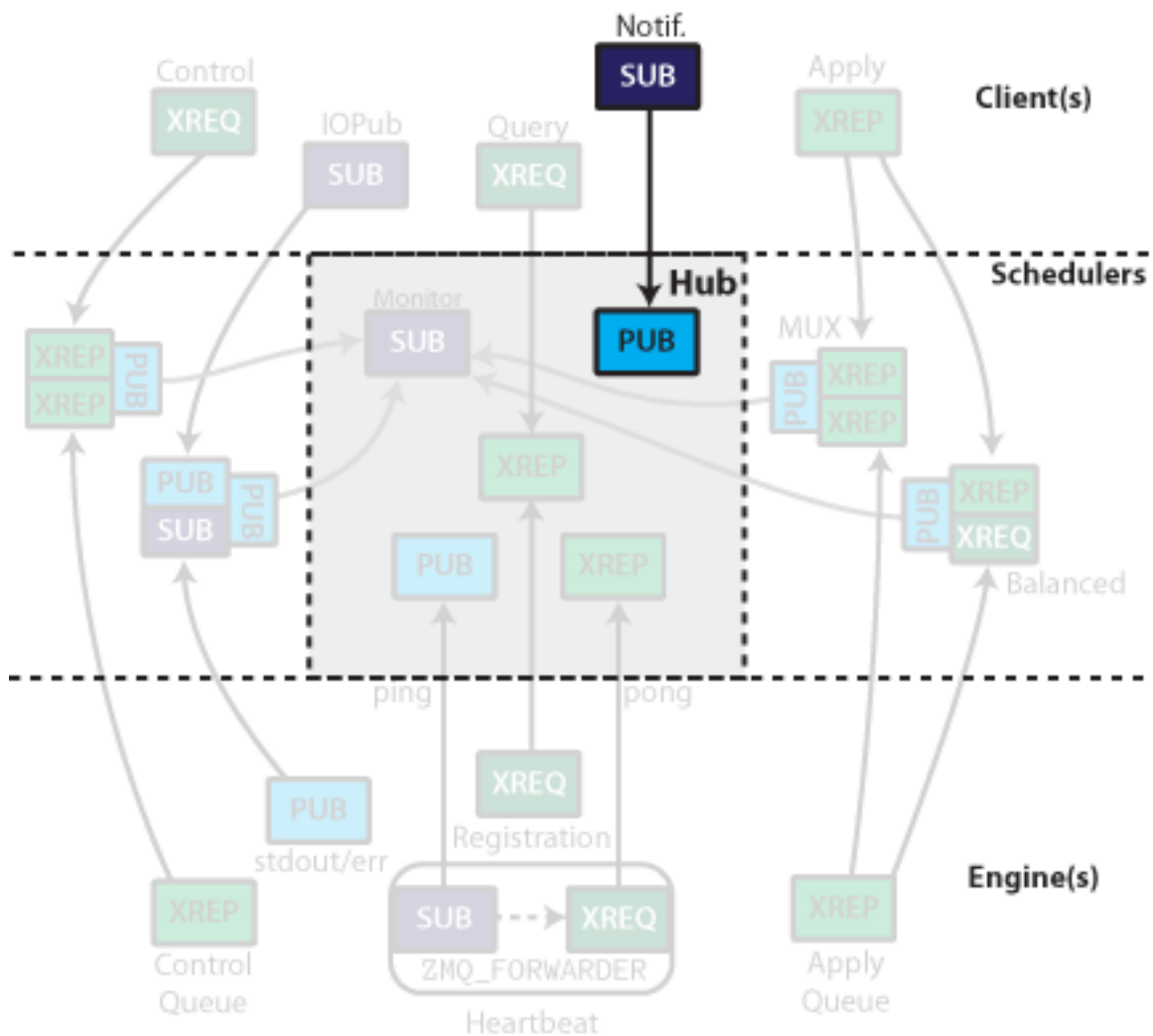


Figure 7.7: Engine registration events are published via a PUB socket.

## 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):
```

### 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 `STDOUT_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.

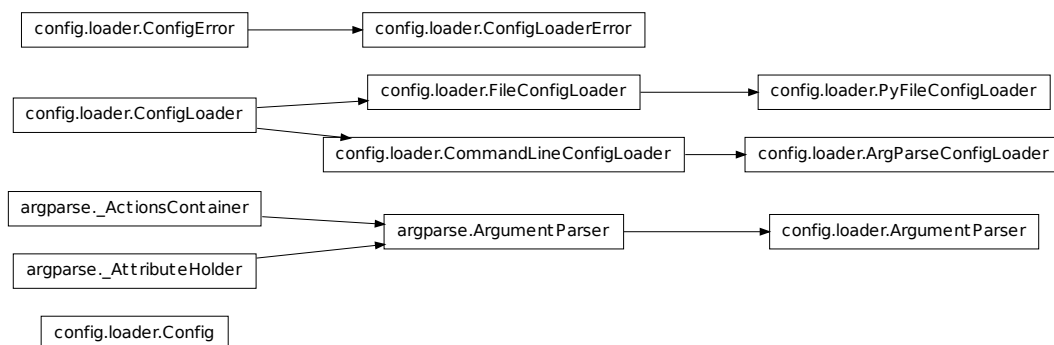
The crash recovery system, is a IPython kernel module for when it fails unexpectedly, you can retrieve the information from the section, this will be based on a log and a lock file to indicate when the kernel was not closed in a proper way.

# 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-
                                         scription=None, epilog=None,
                                         version=None, parents=[
                                         ], formatter_class=<class
                                         'argparse.HelpFormatter'>,
                                         prefix_chars='-', from-
                                         file_prefix_chars=None, ar-
                                         gument_default=None, con-
                                         flict_handler='error', add_help=True)
```

Bases: `argparse.ArgumentParser`

Simple argparse subclass that prints help to stdout by default.

```

__init__ (prog=None, usage=None, description=None, epilog=None, version=None, parents=[], formatter_class=<class 'argparse.HelpFormatter'>, prefix_chars='-', fromfile_prefix_chars=None, argument_default=None, conflict_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)

convert_arg_line_to_args (arg_line)

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

**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**

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** ()

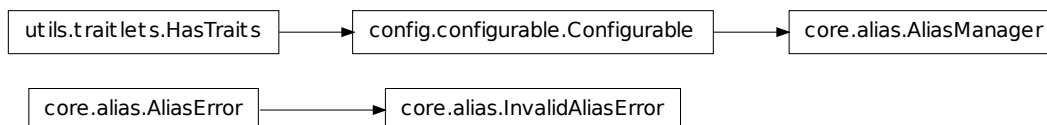
**load\_config** ()

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 uninstall 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

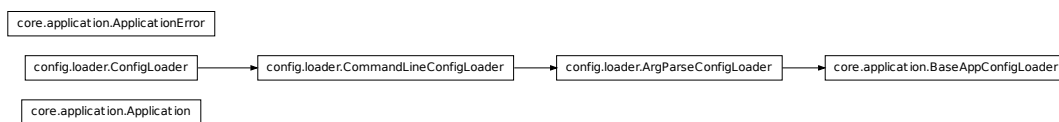
IPython.core.alias.default\_aliases ()

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 componenets.

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 or ~/.config/ipython/

**load\_command\_line\_config()**

Load the command line config.

**load\_file\_config(suppress\_errors=True)**

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**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]
```

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.

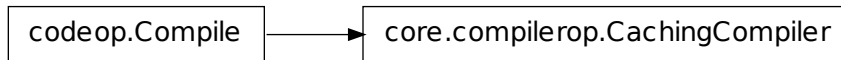
The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

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`:



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
- Thomas Kluyver

### 8.6.2 CachingCompiler

**class** IPython.core.compilerop.CachingCompiler

Bases: codeop.Compile

A compiler that caches code compiled from interactive statements.

**\_\_init\_\_** ()

**cache** (code, number=0)

Make a name for a block of code, and cache the code.

**Parameters** code : str

The Python source code to cache.

**number** : int

A number which forms part of the code's name. Used for the execution counter.

**Returns** The name of the cached code (as a string). Pass this as the filename :

argument to compilation, so that tracebacks are correctly hooked up. :

**check\_cache** (\*args)

Call linecache.checkcache() safely protecting our cached values.

**compiler\_flags**

Flags currently active in the compilation process.

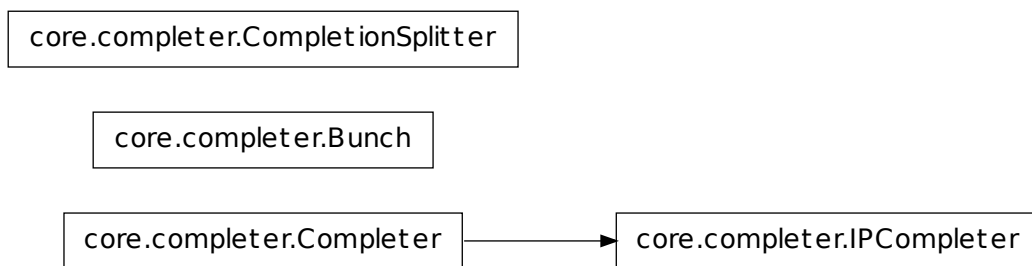
`IPython.core.compilerop.code_name (code, number=0)`  
 Compute a (probably) unique name for code for caching.

This now expects code to be unicode.

## 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*)

Wrapper around the complete method for the benefit of emacs and pydb.

**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=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`:

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.**

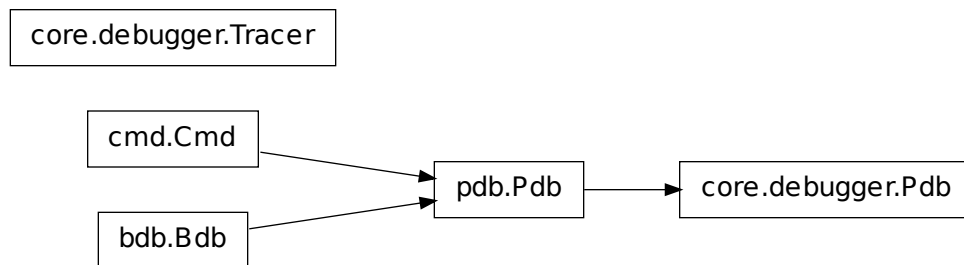
**make\_report** (*traceback*)

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\_is()**

**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 detection 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 exclue 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.

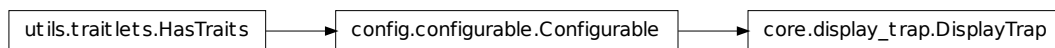
**Parameters** `objs` : tuple of objects

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** ()

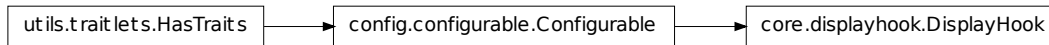
Unset the hook.

## 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, col-
                                         ors='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', out-
         put_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 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** (*format\_dict*)

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.

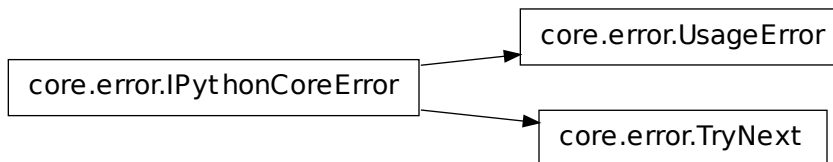
**metadata** : dict

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 'NoColor' and fields for exception handling filled in.

Examples:

```
>>> ec = exception_colors()
>>> ec.active_scheme_name
''
>>> print ec.active_colors
None
```

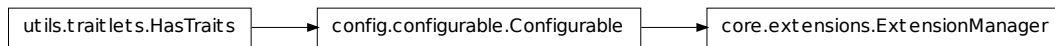
Now we activate a color scheme: >>> ec.set\_active\_scheme('NoColor') >>> ec.active\_scheme\_name 'NoColor' >>> ec.active\_colors.keys() ['em', 'filenameEm', 'excName', 'valEm', 'nameEm', 'line', 'topline', 'name', 'caret', 'val', 'vName', 'Normal', 'filename', 'linenoEm', 'lineno', 'normalEm']



## 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 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.

**`__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*)

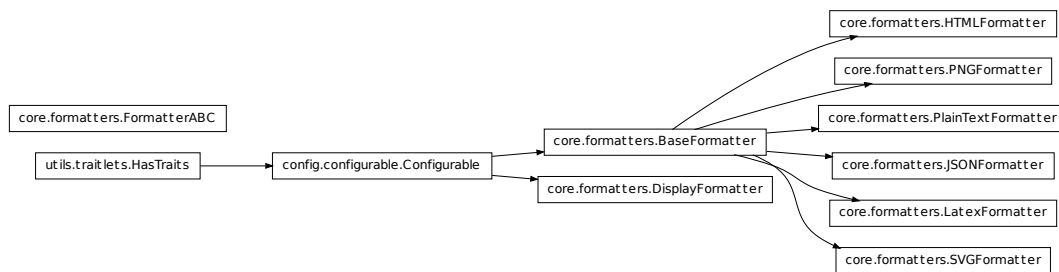
Unload an IPython extension by its module name.

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 traited `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 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.

**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 structure 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 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.

## 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 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.

## 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 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.

## 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 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.

## 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.end_group(7, '])')
```

```
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.

**float\_format**

A trait for strings.

**float\_precision**

A casting version of the string 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 uninstalls 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.

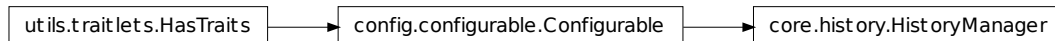
**exclude** : list or tuple, optional

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 Class

#### 8.19.3 `HistoryManager`

**class** `IPython.core.history.HistoryManager` (*shell*, *config=None*, *\*\*traits*)

Bases: `IPython.config.configurable.Configurable`

A class to organize all history-related functionality in one place.

**\_\_init\_\_** (*shell, config=None, \*\*traits*)

Create a new history manager associated with a shell instance.

**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**

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\_cache\_size**

A integer trait.

**db\_input\_cache**

An instance of a Python list.

**db\_log\_output**

A boolean (True, False) trait.

**db\_output\_cache**

An instance of a Python list.

**dir\_hist**

An instance of a Python list.

**end\_session** ()

Close the database session, filling in the end time and line count.

**get\_range** (*session=0, start=1, stop=None, raw=True, output=False*)

Retrieve input by session.

**Parameters** **session** : int

Session number to retrieve. The current session is 0, and negative numbers count back from current session, so -1 is previous session.

**start** : int

First line to retrieve.

**stop** : int

End of line range (excluded from output itself). If None, retrieve to the end of the session.

**raw** : bool

If True, return untranslated input

**output** : bool

If True, attempt to include output. This will be 'real' Python objects for the current session, or text reprs from previous sessions if `db_log_output` was enabled at the time. Where no output is found, None is used.



**Returns** An iterator over the desired lines. Each line is a 3-tuple, either :

(session, line, input) if output is False, or :

(session, line, (input, output)) if output is True. :

**get\_range\_by\_str** (rangestr, raw=True, output=False)

Get lines of history from a string of ranges, as used by magic commands %hist, %save, %macro, etc.

**Parameters** rangestr : str

A string specifying ranges, e.g. “5 ~2/1-4”. See `magic_history()` for full details.

**raw, output** : bool

As `get_range()`

**Returns** Tuples as :meth:‘get\_range’ :

**get\_tail** (n=10, raw=True, output=False, include\_latest=False)

Get the last n lines from the history database.

**Parameters** n : int

The number of lines to get

**raw, output** : bool

See `get_range()`

**include\_latest** : bool

If False (default), n+1 lines are fetched, and the latest one is discarded. This is intended to be used where the function is called by a user command, which it should not return.

**Returns** Tuples as :meth:‘get\_range’ :

**hist\_file**

A trait for unicode strings.

**init\_db** ()

Connect to the database, and create tables if necessary.

**input\_hist\_parsed**

An instance of a Python list.

**input\_hist\_raw**

An instance of a Python list.

**name\_session** (name)

Give the current session a name in the history database.

**new\_session** ()

Get a new session number.

**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.

**output\_hist**

An instance of a Python dict.

**output\_hist\_reprs**

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.

**reset** (*new\_session=True*)

Clear the session history, releasing all object references, and optionally open a new session.

**search** (*pattern='\*', raw=True, search\_raw=True, output=False*)

Search the database using unix glob-style matching (wildcards \* and ?).

**Parameters** **pattern** : str

The wildcarded pattern to match when searching

**search\_raw** : bool

If True, search the raw input, otherwise, the parsed input

**raw, output** : bool

See `get_range()`

**Returns** Tuples as :meth:‘get\_range’ :

**session\_number**

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.

**store\_inputs** (*line\_num*, *source*, *source\_raw=None*)

Store source and raw input in history and create input cache variables *\_i\**.

**Parameters** *line\_num* : int

The prompt number of this input.

**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.

**store\_output** (*line\_num*)

If database output logging is enabled, this saves all the outputs from the indicated prompt number to the database. It's called by *run\_cell* after code has been executed.

**Parameters** *line\_num* : int

The line number from which to save outputs

**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.

**writeout\_cache** ()

Write any entries in the cache to the database.

## 8.19.4 Functions

*IPython.core.history*.**extract\_hist\_ranges** (*ranges\_str*)

Turn a string of history ranges into 3-tuples of (session, start, stop).

### Examples

```
list(extract_input_ranges("~8/5~7/4 2")) [(-8, 5, None), (-7, 1, 4), (0, 2, 3)]
```

*IPython.core.history*.**init\_ipython** (*ip*)

`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. Use `-n` to show them.

Ranges of history can be indicated using the syntax: `4` : Line 4, current session  
`4-6` : Lines 4-6, current session  
`243/1-5` : Lines 1-5, session 243  
`~2/7` : Line 7, session 2 before current  
`~8/1~6/5` : From the first line of 8 sessions ago, to the fifth line

of 6 sessions ago.

Multiple ranges can be entered, separated by spaces

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

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 saved history (almost all commands ever written). Use `'%hist -g'` to show full saved history (may be very long).

- l: get the last `n` lines from all sessions. Specify `n` as a single arg, or the default is the last 10 lines.

- 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.

## Examples

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

`IPython.core.history.magic_rep(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:

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

`%rep 45`

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

`%rep 1-4`

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

`%rep foo+bar`

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

`IPython.core.history.magic_rerun(self, parameter_s='')`

Re-run previous input

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

Options:

- l <n> : Repeat the last n lines of input, not including the current command.
- g foo : Repeat the most recent line which contains foo

## 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 line number.

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, line number, 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

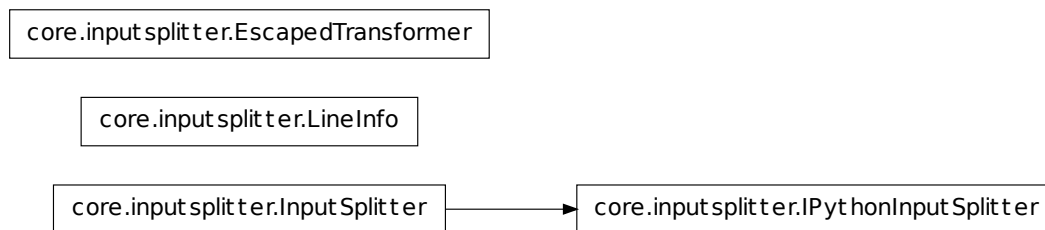
Typically, shutdown hooks should raise `TryNext` so all shutdown ops are done

`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.

## 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.

### 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_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.

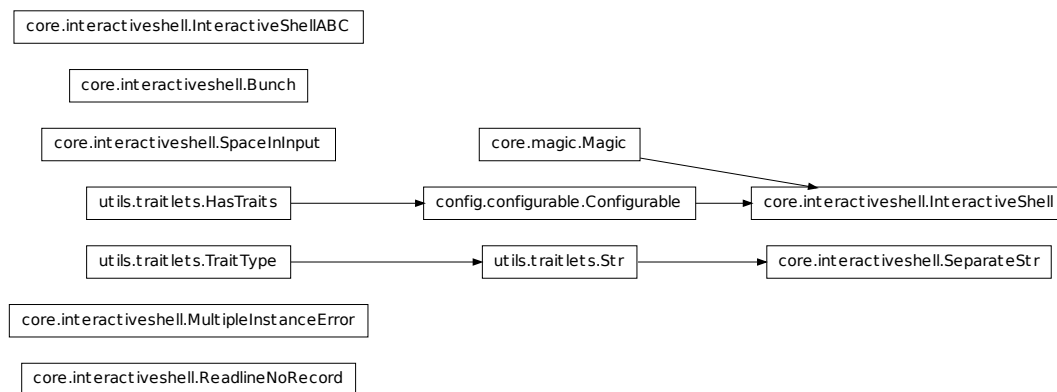
`IPython.core.inputsplitter.transform_ipy_prompt(line)`

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.

**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\_lines** (*range\_str, raw=False*)

Return as a string a set of input history slices.

Inputs:

- range\_str*: the set of slices is given as a string, like

“~5/6~4/2 4:8 9”, since this function is for use by magic functions which get their arguments as strings. The number before the / is the session number: ~n goes n back from the current session.

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 -&gt; standard python form, means including items N...(M-1).

N-M -&gt; include items N..M (closed endpoint).

**filename**

A trait for unicode strings.

**find\_user\_code** (*target*, *raw=True*)

Get a code string from history, file, or a string or macro.

This is mainly used by magic functions.

**Parameters** **target** : str

A string specifying code to retrieve. This will be tried respectively as: ranges of input history (see %history for syntax), a filename, or an expression evaluating to a string or Macro in the user namespace.

**raw** : bool

If true (default), retrieve raw history. Has no effect on the other retrieval mechanisms.

**Returns** A string of code. :

**ValueError** is raised if nothing is found, and **TypeError** if it evaluates :

to an object of another type. In each case, **.args[0]** is a printable :

**message.** :

**format\_latex** (*string*)

Format a string for latex inclusion.

**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.

**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), programmatically (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 unicode strings.

**logfile**

A trait for unicode 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'.

## Examples

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

**`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.

## Examples

To get a plain black and white terminal:

```
%colors nocolor
```

**`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:

- If the argument is a filename, IPython will load that into the editor. It will execute its contents with `execfile()` when you exit, loading any code in the file into your interactive namespace.

- The arguments are ranges of input history, e.g. “7 ~1/4-6”.

The syntax is the same as in the `%history` magic.

- If the argument is a string variable, its contents are loaded into the editor. You can thus edit any string which contains python code (including the result of previous edits).

- If the argument is the name of an object (other than a string),

IPython will try to locate the file where it was defined and open the editor at the point where it is defined. You can use `%edit function` to load an editor exactly at the point where ‘function’ is defined, edit it and have the file be executed automatically.

If the object is a macro (see `%macro` for details), this opens up your specified editor with a temporary file containing the macro’s data. Upon exit, the macro is reloaded with the contents of the file.

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

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

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

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

```
In [1]: 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 macro for future re-execution. It accepts ranges of history, filenames or string objects.

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

Options:

-r: use 'raw' input. By default, the 'processed' history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed 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 syntax for indicating input ranges is described in %history.

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

For example, if your history contains (%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'.
```

**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\_pastein** (*parameter\_s=''*)

Upload code to the 'Lodge it' paste bin, returning the URL.

**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.

## Examples

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

**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\_precision** (*s*='')

Set floating point precision for pretty printing.

Can set either integer precision or a format string.

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

If no argument is given, defaults will be restored.

## Examples

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

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

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

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

```
In [5]: pi
```



```
Out [5]: 3
```

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

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

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

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

**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.

## Examples

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

**`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', 'osx' 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_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.

**Parameters** **-f** : force reset without asking for confirmation.

**-s** : ‘Soft’ reset: Only clears your namespace, leaving history intact. References to objects may be kept. By default (without this option), we do a ‘hard’ reset, giving you a new session and removing all references to objects from the current session.

### Examples

In [6]: a = 1

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

In [8]: ‘a’ in \_ip.user\_ns Out[8]: True

In [9]: %reset -f

In [1]: ‘a’ in \_ip.user\_ns Out[1]: False

**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 0x105496d70>*)

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 or a macro to a given filename.

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

Options:

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

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

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

**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.

## Examples

Define two variables and list them with `who`:

```
In [1]: alpha = 123
```

```
In [2]: beta = 'test'
```

```
In [3]: %who
alpha    beta
```

```
In [4]: %who int
alpha
```

```
In [5]: %who str
beta
```

**`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.

### Examples

Define two variables and list them with `who_ls`:

```
In [1]: alpha = 123
```

```
In [2]: beta = 'test'
```

```
In [3]: %who_ls
```

```
Out[3]: ['alpha', 'beta']
```

```
In [4]: %who_ls int
```

```
Out[4]: ['alpha']
```

```
In [5]: %who_ls str
```

```
Out[5]: ['beta']
```

**`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 arrays, a summary with shape, number of elements, typecode and size in memory.
- Everything else: a string representation, snipping their middle if too long.

### Examples

Define two variables and list them with `whos`:

```
In [1]: alpha = 123
```

```
In [2]: beta = 'test'
```

```
In [3]: %whos
```

Variable	Type	Data/Info
----------	------	-----------

-----

```
alpha      int      123
beta       str      test
```

**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 unicode 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.

**refill\_readline\_hist** ()

**register\_post\_execute** (*func*)

Register a function for calling after code execution.

**reset** (*new\_session=True*)

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.

If `new_session` is `True`, a new history session will be opened.

**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\_ast\_nodes** (*odelist, cell\_name, interactivity='last'*)

Run a sequence of AST nodes. The execution mode depends on the interactivity parameter.

**Parameters** **odelist** : list

A sequence of AST nodes to run.

**cell\_name** : str

Will be passed to the compiler as the filename of the cell. Typically the value returned by `ip.compile.cache(cell)`.

**interactivity** : str

'all', 'last' or 'none', specifying which nodes should be run interactively (displaying output from expressions). Other values for this parameter will raise a `ValueError`.



**run\_cell** (*cell*, *store\_history=True*)

Run a complete IPython cell.

**Parameters** *cell* : str

The code (including IPython code such as %magic functions) to run.

**store\_history** : bool

If True, the raw and translated cell will be stored in IPython's history. For user code calling back into IPython's machinery, this should be set to False.

**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\_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.

**runcode** (*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\_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**

### ReadlineNoRecord

**class** IPython.core.interactiveshell.**ReadlineNoRecord** (*shell*)

Bases: object

Context manager to execute some code, then reload readline history so that interactive input to the code doesn't appear when pressing up.

**\_\_init\_\_** (*shell*)

**current\_length** ()

**get\_readline\_tail** (*n=10*)

Get the last n items in readline history.

### SeparateStr

**class** IPython.core.interactiveshell.**SeparateStr** (*default\_value=<IPython.utils.traitlets.NoDefaultSpecified object at 0x1054d4950>*,  
\*\**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
0x1054d4950>, **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_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`:

core.macro.Macro

Support for interactive macros in IPython

### 8.25.2 Macro

**class** `IPython.core.macro.Macro` (*code*)

Bases: `object`

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\_\_** (*code*)

store the macro value, as a single string which can be executed

## 8.26 core.magic

### 8.26.1 Module: `core.magic`

Inheritance diagram for `IPython.core.magic`:

core.magic.Bunch

core.magic.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\_lines** (*range\_str, raw=False*)

Return as a string a set of input history slices.

Inputs:

- `range_str`: the set of slices is given as a string, like

“~5/6-~4/2 4:8 9”, since this function is for use by magic functions which get their arguments as strings. The number before the `/` is the session number: `~n` goes `n` back from the current session.

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'.

## Examples

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

**`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.

## Examples

To get a plain black and white terminal:

```
%colors nocolor
```

**`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:

- If the argument is a filename, IPython will load that into the editor. It will execute its contents with `execfile()` when you exit, loading any code in the file into your interactive namespace.

- The arguments are ranges of input history, e.g. "7 ~1/4-6".

The syntax is the same as in the `%history` magic.

- If the argument is a string variable, its contents are loaded into the editor. You can thus edit any string which contains python code (including the result of previous edits).

- If the argument is the name of an object (other than a string),

IPython will try to locate the file where it was defined and open the editor at the point where it is defined. You can use `%edit function` to load an editor exactly at the point where 'function' is defined, edit it and have the file be executed automatically.

If the object is a macro (see `%macro` for details), this opens up your specified editor with a temporary file containing the macro's data. Upon exit, the macro is reloaded with the contents of the file.

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

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

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



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

```
In [1]: 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 macro for future re-execution. It accepts ranges of history, filenames or string objects.

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

Options:

-r: use 'raw' input. By default, the 'processed' history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed 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 syntax for indicating input ranges is described in %history.

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

For example, if your history contains (%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'.
```

**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\_pastebin** (*parameter\_s*='')

Upload code to the 'Lodge it' paste bin, returning the URL.

**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.

## Examples

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

**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\_precision** (*s*='')

Set floating point precision for pretty printing.

Can set either integer precision or a format string.

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

If no argument is given, defaults will be restored.

## Examples

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

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

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

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

```
In [5]: pi
```

```
Out [5]: 3
```

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

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

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

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

**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.



## Examples

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

**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', 'osx' 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\_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.

**Parameters** **-f** : force reset without asking for confirmation.

**-s** : ‘Soft’ reset: Only clears your namespace, leaving history intact. References to objects may be kept. By default (without this option), we do a ‘hard’ reset, giving you a new session and removing all references to objects from the current session.

### Examples

```
In [6]: a = 1
```

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

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

```
In [9]: %reset -f
```

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

**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 0x105496d70>*)  
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 or a macro to a given filename.

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

Options:

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

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

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

**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.

## Examples

Define two variables and list them with `who`:

In [1]: `alpha = 123`

In [2]: `beta = 'test'`

In [3]: `%who`  
`alpha beta`

In [4]: `%who int`  
`alpha`



```
In [5]: %who str
beta
```

**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.

### Examples

Define two variables and list them with who\_ls:

```
In [1]: alpha = 123
```

```
In [2]: beta = 'test'
```

```
In [3]: %who_ls
```

```
Out[3]: ['alpha', 'beta']
```

```
In [4]: %who_ls int
```

```
Out[4]: ['alpha']
```

```
In [5]: %who_ls str
```

```
Out[5]: ['beta']
```

**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 arrays, a summary with shape, number of elements, typecode and size in memory.
- Everything else: a string representation, snipping their middle if too long.

### Examples

Define two variables and list them with whos:

```
In [1]: alpha = 123
```

```
In [2]: beta = 'test'
```

```
In [3]: %whos
```

Variable	Type	Data/Info
-----		

```
alpha      int      123
beta       str      test
```

**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.needs_local_scope` (*func*)

Decorator to mark magic functions which need to local scope to run.

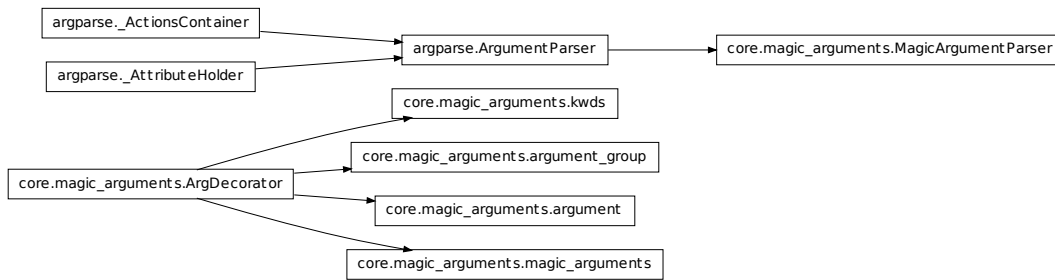
`IPython.core.magic.on_off` (*tag*)

Return an ON/OFF string for a 1/0 input. Simple utility function.

## 8.27 core.magic\_arguments

### 8.27.1 Module: core.magic\_arguments

Inheritance diagram for `IPython.core.magic_arguments`:



A decorator-based method of constructing IPython magics with *argparse* option handling.

New magic functions can be defined like so:

```

from IPython.core.magic_arguments import (argument, magic_arguments,
    parse_argstring)

@magic_arguments()
@argument('-o', '--option', help='An optional argument.')
@argument('arg', type=int, help='An integer positional argument.')
def magic_cool(self, arg):
    """ A really cool magic command.

    """
    args = parse_argstring(magic_cool, arg)
    ...
  
```

The `@magic_arguments` decorator marks the function as having *argparse* arguments. The `@argument` decorator adds an argument using the same syntax as *argparse*'s `add_argument()` method. More sophisticated uses may also require the `@argument_group` or `@kwds` decorator to customize the formatting and the parsing.

Help text for the magic is automatically generated from the docstring and the arguments:

```

In[1]: %cool?
%cool [-o OPTION] arg

A really cool magic command.

positional arguments:
  arg                An integer positional argument.

optional arguments:
  -o OPTION, --option OPTION
                        An optional argument.
  
```

## 8.27.2 Classes

### ArgDecorator

**class** IPython.core.magic\_arguments.ArgDecorator

Bases: object

Base class for decorators to add ArgumentParser information to a method.

**\_\_init\_\_** ()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**add\_to\_parser** (parser, group)

Add this object's information to the parser, if necessary.

### MagicArgumentParser

**class** IPython.core.magic\_arguments.MagicArgumentParser (prog=None, usage=None, description=None, epilog=None, version=None, parents=None, formatter\_class=<class 'argparse.HelpFormatter'>, prefix\_chars='-', argument\_default=None, conflict\_handler='error', add\_help=False)

Bases: argparse.ArgumentParser

An ArgumentParser tweaked for use by IPython magics.

**\_\_init\_\_** (prog=None, usage=None, description=None, epilog=None, version=None, parents=None, formatter\_class=<class 'argparse.HelpFormatter'>, prefix\_chars='-', argument\_default=None, conflict\_handler='error', add\_help=False)

**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)

**convert\_arg\_line\_to\_args** (arg\_line)

**error** (message)

Raise a catchable error instead of exiting.

```
exit (status=0, message=None)  
format_help ()  
format_usage ()  
format_version ()  
get_default (dest)  
parse_args (args=None, namespace=None)  
parse_argstring (argstring)  
    Split a string into an argument list and parse that argument list.  
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)
```

## argument

```
class IPython.core.magic_arguments.argument (*args, **kws)  
    Bases: IPython.core.magic_arguments.ArgDecorator  
  
    Store arguments and keywords to pass to add_argument().  
  
    Instances also serve to decorate command methods.  
  
    __init__ (*args, **kws)  
  
    add_to_parser (parser, group)  
        Add this object's information to the parser.
```

## argument\_group

```
class IPython.core.magic_arguments.argument_group (*args, **kws)  
    Bases: IPython.core.magic_arguments.ArgDecorator  
  
    Store arguments and keywords to pass to add_argument_group().  
  
    Instances also serve to decorate command methods.  
  
    __init__ (*args, **kws)  
  
    add_to_parser (parser, group)  
        Add this object's information to the parser.
```

### **kwds**

```
class IPython.core.magic_arguments.kwds (**kwds)
    Bases: IPython.core.magic_arguments.ArgDecorator

    Provide other keywords to the sub-parser constructor.

    __init__ (**kwds)

    add_to_parser (parser, group)
        Add this object's information to the parser, if necessary.
```

### **magic\_arguments**

```
class IPython.core.magic_arguments.magic_arguments (name=None)
    Bases: IPython.core.magic_arguments.ArgDecorator

    Mark the magic as having argparse arguments and possibly adjust the name.

    __init__ (name=None)

    add_to_parser (parser, group)
        Add this object's information to the parser, if necessary.
```

## **8.27.3 Functions**

```
IPython.core.magic_arguments.construct_parser (magic_func)
    Construct an argument parser using the function decorations.

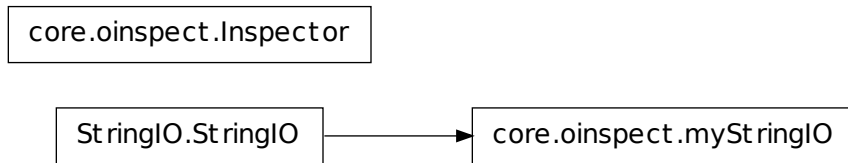
IPython.core.magic_arguments.parse_argstring (magic_func, argstring)
    Parse the string of arguments for the given magic function.

IPython.core.magic_arguments.real_name (magic_func)
    Find the real name of the magic.
```

## **8.28 core.oinspect**

### **8.28.1 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.

## 8.28.2 Classes

### Inspector

```

class IPython.core.oinspect.Inspector (color_table='':
    <IPython.utils.coloransi.ColorScheme instance at 0x104554a28>, 'LightBG':
    <IPython.utils.coloransi.ColorScheme instance at 0x104554248>, 'NoColor':
    <IPython.utils.coloransi.ColorScheme instance at 0x104554c20>, 'Linux':
    <IPython.utils.coloransi.ColorScheme instance at 0x104554a28>},
    code_color_table='':
    <IPython.utils.coloransi.ColorScheme instance at 0x1045e7560>, 'LightBG':
    <IPython.utils.coloransi.ColorScheme instance at 0x1045e75a8>, 'NoColor':
    <IPython.utils.coloransi.ColorScheme instance at 0x1045e7098>, 'Linux':
    <IPython.utils.coloransi.ColorScheme instance at 0x1045e7560>}, scheme='NoColor',
    str_detail_level=0)
  
```

```
__init__ (color_table={'': <IPython.utils.coloransi.ColorScheme instance at 0x104554a28>, 'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x104554248>, 'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x104554c20>, 'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x104554a28>}, code_color_table={'': <IPython.utils.coloransi.ColorScheme instance at 0x1045e7560>, 'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x1045e75a8>, 'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x1045e7098>, 'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x1045e7560>}, 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(' ')`

### 8.28.3 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.

## 8.29 core.page

### 8.29.1 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. —

### 8.29.2 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  $\leq 0$ , `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.

## 8.30 core.payload

### 8.30.1 Module: `core.payload`

Inheritance diagram for `IPython.core.payload`:



Payload system for IPython.

Authors:

- Fernando Perez
- Brian Granger

### 8.30.2 `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 uninstall 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*)

## 8.31 core.payloadpage

### 8.31.1 Module: core.payloadpage

A payload based version of page.

Authors:

- Brian Granger
- Fernando Perez

### 8.31.2 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.

## 8.32 core.plugin

### 8.32.1 Module: `core.plugin`

Inheritance diagram for `IPython.core.plugin`:



IPython plugins.

Authors:

- Brian Granger

### 8.32.2 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 uninstall 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 uninstall 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*)

## 8.33 core.prefilter

### 8.33.1 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

## 8.33.2 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 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.

### **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 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.

**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,          pre-
                                           filter_manager=None,    con-
                                           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 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.

## 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,          pre-
                                             filter_manager=None,   con-
                                             fig=None)
```

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, filter_manager=None, 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.

**IPyPromptTransformer**

```
class IPython.core.prefilter.IPyPromptTransformer (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.

## MacroChecker

```
class IPython.core.prefilter.MacroChecker (shell=None, prefilter_manager=None,
                                           config=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 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.

## MacroHandler

```
class IPython.core.prefilter.MacroHandler (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*)

**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.

**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, filter\_manager=None, fig=None*) *pre-con-*

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 `pyparsing` 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 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.

**ShellEscapeHandler**

```
class IPython.core.prefilter.ShellEscapeHandler (shell=None,           pre-
                                              filter_manager=None,      con-
                                              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.

### 8.33.3 Function

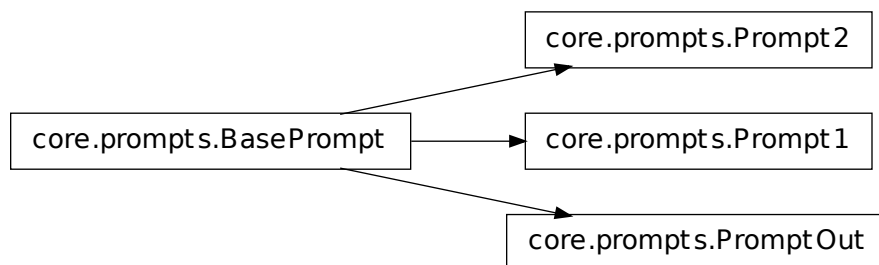
`IPython.core.prefilter.is_shadowed(identifier, ip)`

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.

## 8.34 core.prompts

### 8.34.1 Module: `core.prompts`

Inheritance diagram for `IPython.core.prompts`:



Classes for handling input/output prompts.

Authors:

- Fernando Perez
- Brian Granger



## 8.34.2 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 [N#]: ', pad\_left=True*)

Bases: IPython.core.prompts.BasePrompt

Input interactive prompt similar to Mathematica's.

**\_\_init\_\_** (*cache, sep='n', prompt='In [N#]: ', 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*)

### 8.34.3 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.

## 8.35 core.splitinput

### 8.35.1 Module: core.splitinput

Simple utility for splitting user input.

Authors:

- Brian Granger
- Fernando Perez

`IPython.core.splitinput.split_user_input (line, pattern=None)`

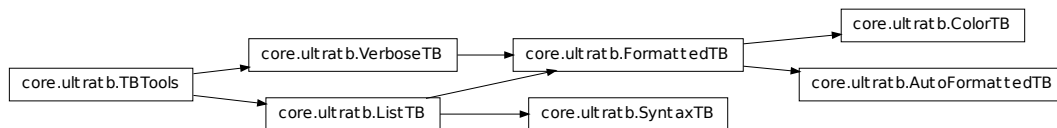
Split user input into pre-char/whitespace, function part and rest.

This is currently handles lines with '=' in them in a very inconsistent manner.

## 8.36 core.ultratb

### 8.36.1 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.

## 8.36.2 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, 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=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, 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** ()

## 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, 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** ()

**ListTB**

```
class IPython.core.ultratb.ListTB (color_scheme='NoColor',    call_pdb=False,    os-  
                                     tream=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, 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.

## 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, 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.

## 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, evalute, 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.

### 8.36.3 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()`

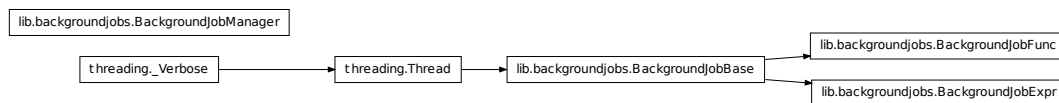
Print a message about internal inspect errors.

These are unfortunately quite common.

## 8.37 lib.backgroundjobs

### 8.37.1 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).

### 8.37.2 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 `strform` 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*)

## 8.38 lib.clipboard

### 8.38.1 Module: `lib.clipboard`

Utilities for accessing the platform's clipboard.

### 8.38.2 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.

## 8.39 lib.deepreload

### 8.39.1 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.

### 8.39.2 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)`

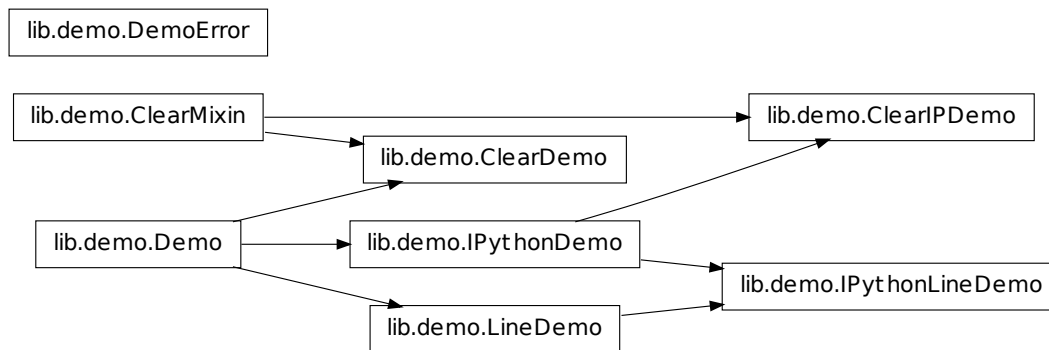
`IPython.lib.deepreload.reload(module, exclude=['sys', '__builtin__', '__main__'])`

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.

## 8.40 lib.demo

### 8.40.1 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>
#####
```

## 8.40.2 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

### 8.40.3 Function

`IPython.lib.demo.re_mark(mark)`

## 8.41 lib.guisupport

### 8.41.1 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.

### 8.41.2 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.

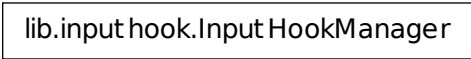
`IPython.lib.guisupport.start_event_loop_wx(app=None)`

Start the wx event loop in a consistent manner.

## 8.42 lib.inputhook

### 8.42.1 Module: `lib.inputhook`

Inheritance diagram for `IPython.lib.inputhook`:



```
graph TD; A[lib.inputhook.InputHookManager];
```

Inputhook management for GUI event loop integration.

### 8.42.2 `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 a 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\_inpthook** ()

Return the current `PyOS_InputHook` as a `ctypes.c_void_p`.

**get\_pyos\_inpthook\_as\_func** ()

Return the current `PyOS_InputHook` as a `ctypes.PYFUNCTYPE`.

**set\_inpthook** (*callback*)

Set `PyOS_InputHook` to `callback` and return the previous one.

`IPython.lib.inputhook.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. :**

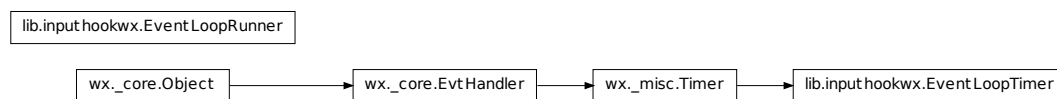
## 8.43 lib.inputhookgtk

### 8.43.1 Module: lib.inputhookgtk

## 8.44 lib.inputhookwx

### 8.44.1 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

### 8.44.2 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

### 8.44.3 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.

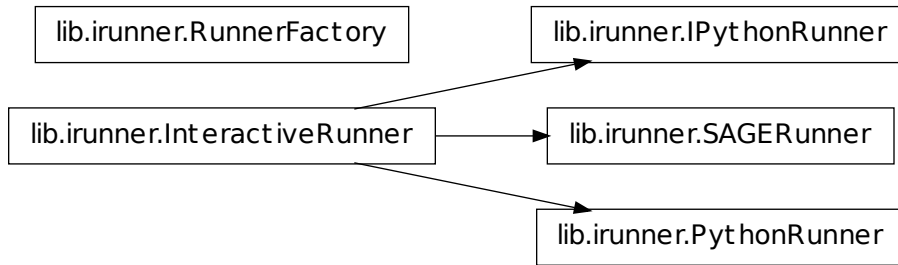
This is like `inputhook_wx1`, but it keeps processing pending events until stdin is ready. After processing all pending events, a call to `time.sleep` is inserted. This is needed, otherwise, CPU usage is at 100%. This sleep time should be tuned though for best performance.

`IPython.lib.inputhookwx.stdin_ready()`

## 8.45 lib.irunner

### 8.45.1 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.

## 8.45.2 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.

### 8.45.3 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`.

## 8.46 lib.latextools

### 8.46.1 Module: `lib.latextools`

Tools for handling LaTeX.

Authors:

- Brian Granger



## 8.46.2 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.

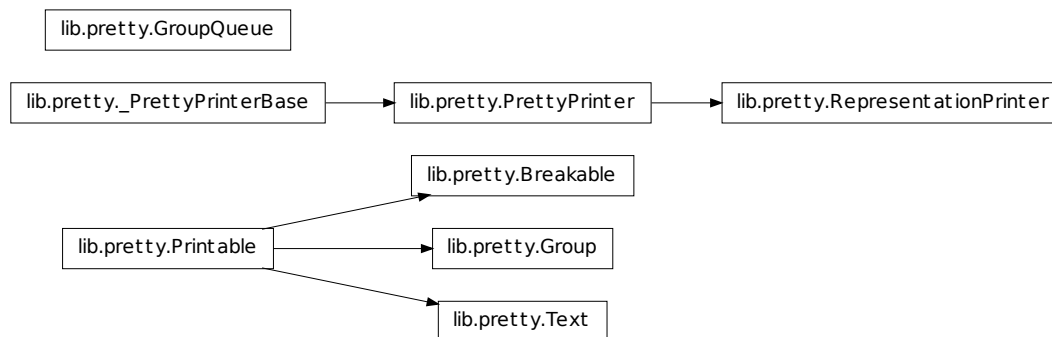
`encode` : bool, optional

Should the PNG data bebase64 encoded to make it JSON'able.

## 8.47 lib.pretty

### 8.47.1 Module: lib.pretty

Inheritance diagram for `IPython.lib.pretty`:



pretty ~~

Python advanced pretty printer. This pretty printer is intended to replace the old *pprint* python module which does not allow developers to provide their own pretty print callbacks.

This module is based on ruby's *prettyprint.rb* library by *Tanaka Akira*.

## Example Usage

To directly print the representation of an object use *pprint*:

```
from pretty import pprint
pprint(complex_object)
```

To get a string of the output use *pretty*:

```
from pretty import pretty
string = pretty(complex_object)
```

## Extending

The pretty library allows developers to add pretty printing rules for their own objects. This process is straightforward. All you have to do is to add a `__pretty__` method to your object and call the methods on the pretty printer passed:

```
class MyObject(object):

    def __pretty__(self, p, cycle):
        ...
```

Depending on the python version you want to support you have two possibilities. The following list shows the python 2.5 version and the compatibility one.

Here the example implementation of a `__pretty__` method for a list subclass for python 2.5 and higher (python 2.5 requires the with statement `__future__` import):

```
class MyList(list):

    def __pretty__(self, p, cycle):
        if cycle:
            p.text('MyList(...)')
        else:
            with p.group(8, 'MyList([' , '])'):
                for idx, item in enumerate(self):
                    if idx:
                        p.text(',')
                        p.breakable()
                    p.pretty(item)
```

The *cycle* parameter is *True* if pretty detected a cycle. You *have* to react to that or the result is an infinite loop. *p.text()* just adds non breaking text to the output, *p.breakable()* either adds a whitespace or breaks here. If you pass it an argument it's used instead of the default space. *p.pretty* prettyprints another object using the pretty print method.

The first parameter to the *group* function specifies the extra indentation of the next line. In this example the next item will either be not broken (if the items are short enough) or aligned with the right edge of the opening bracketed of *MyList*.

If you want to support python 2.4 and lower you can use this code:

```
class MyList(list):

    def __pretty__(self, p, cycle):
        if cycle:
            p.text('MyList(...)')
        else:
            p.begin_group(8, 'MyList([')
            for idx, item in enumerate(self):
                if idx:
                    p.text(',')
                    p.breakable()
                p.pretty(item)
            p.end_group(8, '])')
```

If you just want to indent something you can use the group function without open / close parameters. Under python 2.5 you can also use this code:

```
with p.indent(2):
    ...
```

Or under python2.4 you might want to modify `p.indentation` by hand but this is rather ugly.

**copyright** 2007 by Armin Ronacher. Portions (c) 2009 by Robert Kern.

**license** BSD License.

## 8.47.2 Classes

### Breakable

```
class IPython.lib.pretty.Breakable(seq, width, pretty)
    Bases: IPython.lib.pretty.Printable
    __init__(seq, width, pretty)
    output(stream, output_width)
```

### Group

```
class IPython.lib.pretty.Group(depth)
    Bases: IPython.lib.pretty.Printable
    __init__(depth)
    output(stream, output_width)
```

### GroupQueue

```
class IPython.lib.pretty.GroupQueue(*groups)
    Bases: object
```

```
__init__ (*groups)
deq()
enq(group)
remove(group)
```

## PrettyPrinter

**class** IPython.lib.pretty.**PrettyPrinter** (output, max\_width=79, newline='n')

Bases: IPython.lib.pretty.\_PrettyPrinterBase

Baseclass for the *RepresentationPrinter* prettyprinter that is used to generate pretty reprs of objects. Contrary to the *RepresentationPrinter* this printer knows nothing about the default pprinters or the `__pretty__` callback method.

```
__init__ (output, max_width=79, newline='n')
```

**begin\_group** (indent=0, open='')

Begin a group. If you want support for python < 2.5 which doesn't has the with statement this is the preferred way:

```
p.begin_group(1, '{') ... p.end_group(1, '}')
```

The python 2.5 expression would be this:

```
with p.group(1, '{', '}'): ...
```

The first parameter specifies the indentation for the next line (usually the width of the opening text), the second the opening text. All parameters are optional.

**breakable** (sep=' ')

Add a breakable separator to the output. This does not mean that it will automatically break here. If no breaking on this position takes place the *sep* is inserted which default to one space.

**end\_group** (dedent=0, close='')

End a group. See *begin\_group* for more details.

**flush** ()

Flush data that is left in the buffer.

**group** (\*args, \*\*kws)

like *begin\_group* / *end\_group* but for the with statement.

**indent** (\*args, \*\*kws)

with statement support for indenting/dedenting.

**text** (obj)

Add literal text to the output.

## Printable

**class** IPython.lib.pretty.**Printable**

Bases: object

**\_\_init\_\_** ()

x.**\_\_init\_\_**(...) initializes x; see x.**\_\_class\_\_**.**\_\_doc\_\_** for signature

**output** (stream, output\_width)

## RepresentationPrinter

**class** IPython.lib.pretty.**RepresentationPrinter** (output, verbose=False, max\_width=79, newline='n', singleton\_pprinters=None, type\_pprinters=None, deferred\_pprinters=None)

Bases: IPython.lib.pretty.PrettyPrinter

Special pretty printer that has a *pretty* method that calls the pretty printer for a python object.

This class stores processing data on *self* so you must *never* use this class in a threaded environment. Always lock it or reinstanciate it.

Instances also have a verbose flag callbacks can access to control their output. For example the default instance repr prints all attributes and methods that are not prefixed by an underscore if the printer is in verbose mode.

**\_\_init\_\_** (output, verbose=False, max\_width=79, newline='n', singleton\_pprinters=None, type\_pprinters=None, deferred\_pprinters=None)

**begin\_group** (indent=0, open='')

Begin a group. If you want support for python < 2.5 which doesn't has the with statement this is the preferred way:

```
p.begin_group(1, '{') ... p.end_group(1, '}')
```

The python 2.5 expression would be this:

```
with p.group(1, '{', '}'): ...
```

The first parameter specifies the indentation for the next line (usually the width of the opening text), the second the opening text. All parameters are optional.

**breakable** (sep=' ')

Add a breakable separator to the output. This does not mean that it will automatically break here. If no breaking on this position takes place the *sep* is inserted which default to one space.

**end\_group** (dedent=0, close='')

End a group. See *begin\_group* for more details.

**flush** ()

Flush data that is left in the buffer.

**group** (*\*args, \*\*kws*)  
like `begin_group` / `end_group` but for the `with` statement.

**indent** (*\*args, \*\*kws*)  
with statement support for indenting/dedenting.

**pretty** (*obj*)  
Pretty print the given object.

**text** (*obj*)  
Add literal text to the output.

## Text

```
class IPython.lib.pretty.Text
    Bases: IPython.lib.pretty.Printable
    __init__ ()
    add (obj, width)
    output (stream, output_width)
```

### 8.47.3 Functions

`IPython.lib.pretty.for_type` (*typ, func*)  
Add a pretty printer for a given type.

`IPython.lib.pretty.for_type_by_name` (*type\_module, type\_name, func*)  
Add a pretty printer for a type specified by the module and name of a type rather than the type object itself.

`IPython.lib.pretty.pprint` (*obj, verbose=False, max\_width=79, newline='\n'*)  
Like `pretty` but print to stdout.

`IPython.lib.pretty.pretty` (*obj, verbose=False, max\_width=79, newline='\n'*)  
Pretty print the object's representation.

## 8.48 lib.pylabtools

### 8.48.1 Module: lib.pylabtools

Pylab (matplotlib) support utilities.

## Authors

- Fernando Perez.
- Brian Granger

## 8.48.2 Functions

`IPython.lib.pylabtools.activate_matplotlib(backend)`

Activate the given backend and set interactive to True.

`IPython.lib.pylabtools.figure_size(size_x, size_y)`

Set the default figure size to be [size\_x, size\_y].

This is just an easy to remember, convenience wrapper that sets:

```
matplotlib.rcParams['figure.figsize'] = [size_x, size_y]
```

`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.get_figs(*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

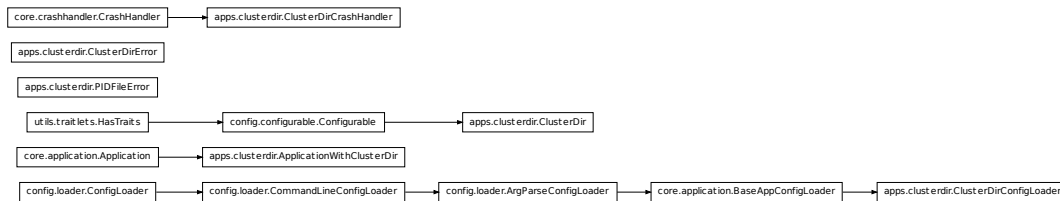
If true, an ‘import **\***’ is done from numpy and pylab.

**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. :

## 8.49 parallel.apps.clusterdir

### 8.49.1 Module: parallel.apps.clusterdir

Inheritance diagram for IPython.parallel.apps.clusterdir:



The IPython cluster directory

### 8.49.2 Classes

#### ApplicationWithClusterDir

**class** IPython.parallel.apps.clusterdir.**ApplicationWithClusterDir** (*argv=None*)  
Bases: IPython.core.application.Application

An application that puts everything into a cluster directory.

Instead of looking for things in the `ipython_dir`, this type of application will use its own private directory called the “cluster directory” for things like config files, log files, etc.

The cluster directory is resolved as follows:

- If the `--cluster-dir` option is given, it is used.
- If `--cluster-dir` is not given, the application directory is resolved using the profile name as `cluster_<profile>`. The search path for this directory is then i) `cwd` if it is found there and ii) in `ipython_dir` otherwise.



The config file for the application is to be put in the cluster dir and named the value of the `config_file_name` class attribute.

**`__init__`** (*argv=None*)

**`attempt`** (*func*)

**`command_line_loader`**

alias of `ClusterDirConfigLoader`

**`construct`** ()

Construct the main objects that make up this app.

**`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`** ()

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(suppress\_errors=True)**

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**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()**

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()**

**start()**

Start the application.

**start\_app()**

Actually start the app.

**start\_logging()**

**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.

## ClusterDir

**class** IPython.parallel.apps.clusterdir.**ClusterDir**(location=u'')

Bases: IPython.config.configurable.Configurable

An object to manage the cluster directory and its resources.

The cluster directory is used by **ipengine**, **ipcontroller** and **ipcluster** to manage the configuration, logging and security of these applications.

This object knows how to find, create and manage these directories. This should be used by any code that want's to handle cluster directories.

**\_\_init\_\_**(location=u'')

**check\_dirs()**

**check\_log\_dir()**

**check\_pid\_dir()**

**check\_security\_dir()**

**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.

**copy\_all\_config\_files** (*path=None, overwrite=False*)

Copy all config files into the active cluster directory.

**copy\_config\_file** (*config\_file, path=None, overwrite=False*)

Copy a default config file into the active cluster directory.

Default configuration files are kept in `IPython.config.default`. This function moves these from that location to the working cluster directory.

**classmethod create\_cluster\_dir** (*csl, cluster\_dir*)

Create a new cluster directory given a full path.

**Parameters** **cluster\_dir** : str

The full path to the cluster directory. If it does exist, it will be used. If not, it will be created.

**classmethod create\_cluster\_dir\_by\_profile** (*path, profile=u'default'*)

Create a cluster dir by profile name and path.

**Parameters** **path** : str

The path (directory) to put the cluster directory in.

**profile** : str

The name of the profile. The name of the cluster directory will be “cluster\_<profile>”.

**classmethod find\_cluster\_dir** (*cluster\_dir*)

Find/create a cluster dir and return its ClusterDir.

This will create the cluster directory if it doesn't exist.

**Parameters** **cluster\_dir** : unicode or str

The path of the cluster directory. This is expanded using `IPython.utils.genutils.expand_path()`.

**classmethod find\_cluster\_dir\_by\_profile** (*ipython\_dir, profile=u'default'*)

Find an existing cluster dir by profile name, return its ClusterDir.

This searches through a sequence of paths for a cluster dir. If it is not found, a `ClusterDirError` exception will be raised.

The search path algorithm is: 1. `os.getcwd()` 2. `ipython_dir` 3. The directories found in the “:” separated

**:env:** **IPCLUSTER\_DIR\_PATH** environment variable.

**Parameters** **ipython\_dir** : unicode or str

The IPython directory to use.

**profile** : unicode or str

The name of the profile. The name of the cluster directory will be “cluster\_<profile>”.

**load\_config\_file** (*filename*)

Load a config file from the top level of the cluster dir.

**Parameters** **filename** : unicode or str

The filename only of the config file that must be located in the top-level of the cluster directory.

**location**

A trait for unicode strings.

**log\_dir**

A trait for unicode strings.

**log\_dir\_name**

A trait for unicode 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.

**pid\_dir**

A trait for unicode strings.

**pid\_dir\_name**

A trait for unicode strings.

**security\_dir**

A trait for unicode strings.

**security\_dir\_name**

A trait for unicode strings.

**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.

### ClusterDirConfigLoader

```
class IPython.parallel.apps.clusterdir.ClusterDirConfigLoader (argv=None,  
                                                                *parser_args,  
                                                                **parser_kw)
```

Bases: `IPython.core.application.BaseAppConfigLoader`

```
__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.

### ClusterDirCrashHandler

```
class IPython.parallel.apps.clusterdir.ClusterDirCrashHandler (app)
```

Bases: `IPython.core.crashhandler.CrashHandler`

sys.excepthook for IPython itself, leaves a detailed report on disk.

```
__init__(app)
make_report(traceback)
    Return a string containing a crash report.
```

### ClusterDirError

```
class IPython.parallel.apps.clusterdir.ClusterDirError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

### PIDFileError

```
class IPython.parallel.apps.clusterdir.PIDFileError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

## 8.50 parallel.apps.ipclusterapp

### 8.50.1 Module: parallel.apps.ipclusterapp

Inheritance diagram for IPython.parallel.apps.ipclusterapp:



The ipcluster application.

## 8.50.2 Classes

### IPClusterApp

```
class IPython.parallel.apps.ipclusterapp.IPClusterApp (argv=None)
    Bases: IPython.parallel.apps.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** (*suppress\_errors=True*)

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**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\_engines()**  
Start the app for the start subcommand.  
**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** (*controller=True*)  
**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.parallel.apps.ipclusterapp.IPClusterAppConfigLoader (argv=None,  
                                                                    *parser_args,  
                                                                    **parser_kw)  
  
Bases: IPython.parallel.apps.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.

### 8.50.3 Function

`IPython.parallel.apps.ipclusterapp.launch_new_instance()`

Create and run the IPython cluster.

## 8.51 parallel.apps.ipcontrollerapp

### 8.51.1 Module: parallel.apps.ipcontrollerapp

Inheritance diagram for `IPython.parallel.apps.ipcontrollerapp`:



The IPython controller application.

## 8.51.2 Classes

### IPControllerApp

```
class IPython.parallel.apps.ipcontrollerapp.IPControllerApp(argv=None)
    Bases: IPython.parallel.apps.clusterdir.ApplicationWithClusterDir
    __init__(argv=None)
    attempt(func)
    command_line_loader
        alias of IPControllerAppConfigLoader
```

```
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()
    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.

**`import_statements()`**

**`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_config_from_json()`**

load config from existing json connector files.

**`load_file_config(suppress_errors=True)`**

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**`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()`**

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`.

**save\_connection\_dict(fname, cdict)**

save a connection dict to json file.

**save\_urls()**

save the registration urls to files.

**set\_command\_line\_config\_log\_level()**

**set\_default\_config\_log\_level()**

**set\_file\_config\_log\_level()**

**start()**

Start the application.

**start\_app()**

**start\_logging()**

**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.

## IPControllerAppConfigLoader

```
class IPython.parallel.apps.ipcontrollerapp.IPControllerAppConfigLoader (argv=None,  
                                                                           *parser_args,  
                                                                           **parser_kw)
```

Bases: `IPython.parallel.apps.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.

### 8.51.3 Function

`IPython.parallel.apps.ipcontrollerapp.launch_new_instance()`

Create and run the IPython controller

## 8.52 parallel.apps.ipengineapp

### 8.52.1 Module: `parallel.apps.ipengineapp`

Inheritance diagram for `IPython.parallel.apps.ipengineapp`:



The IPython engine application

### 8.52.2 Classes

#### **IPEngineApp**

**class** `IPython.parallel.apps.ipengineapp.IEngineApp` (*argv=None*)

Bases: `IPython.parallel.apps.clusterdir.ApplicationWithClusterDir`

**\_\_init\_\_** (*argv=None*)

**attempt** (*func*)

**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** ()

**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** ()

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`.

**find\_url\_file** ()

Set the key file.

Here we don't try to actually see if it exists for is valid as that is hadled by the connection logic.

**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(suppress\_errors=True)**

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**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.

## IPythonAppConfigLoader

```
class IPython.parallel.apps.ipengineapp.IPythonAppConfigLoader (argv=None,
                                                                *parser_args,
                                                                **parser_kw)
```

Bases: `IPython.parallel.apps.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**

### 8.52.3 Function

`IPython.parallel.apps.ipengineapp.launch_new_instance()`

Create and run the IPython controller

## 8.53 parallel.apps.iploggerapp

### 8.53.1 Module: parallel.apps.iploggerapp

Inheritance diagram for `IPython.parallel.apps.iploggerapp`:



A simple IPython logger application

### 8.53.2 Classes

#### IPLoggerApp

**class** `IPython.parallel.apps.iploggerapp.IPLoggerApp` (*argv=None*)

Bases: `IPython.parallel.apps.clusterdir.ApplicationWithClusterDir`

**\_\_init\_\_** (*argv=None*)

**attempt** (*func*)

**command\_line\_loader**

alias of `IPLoggerAppConfigLoader`

**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()**

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** (*suppress\_errors=True*)

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.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the `suppress_errors` option is set to `False`, so errors will make tests fail.

**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()
```

```
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.

## IPLoggerAppConfigLoader

[illegible]

```
__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.

### 8.53.3 Function

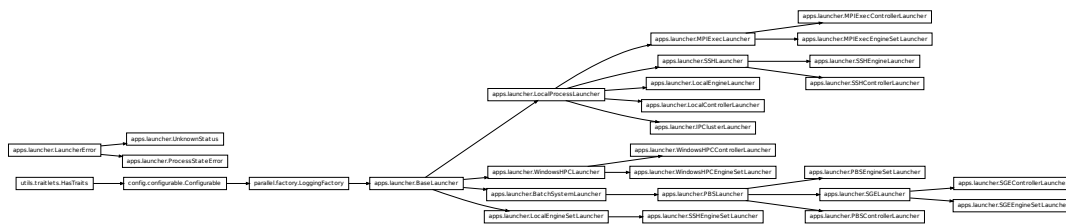
```
IPython.parallel.apps.iploggerapp.launch_new_instance()
```

## Create and run the IPython LogWatcher

## 8.54 parallel.apps.launcher

### 8.54.1 Module: `parallel.apps.launcher`

Inheritance diagram for `IPython.parallel.apps.launcher`:



Facilities for launching IPython processes asynchronously.

### 8.54.2 Classes

#### BaseLauncher

**class** `IPython.parallel.apps.launcher.BaseLauncher` (*work\_dir=u'.'*, *config=None*, *\*\*kwargs*)

Bases: `IPython.parallel.factory.LoggingFactory`

An abstraction for starting, stopping and signaling a process.

**\_\_init\_\_** (*work\_dir=u'.'*, *config=None*, *\*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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\_args()**

The `.args` property calls this to find the args list.

Subcommand should implement this to construct the cmd and args.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int



'KILL', 'INT', etc., or any signal number

**start()**

Start the process.

This must return a deferred that fires with information about the process starting (like a pid, job id, etc.).

**start\_data**

**stop()**

Stop the process and notify observers of stopping.

This must return a deferred that fires with information about the processing stopping, like errors that occur while the process is attempting to be shut down. This deferred won't fire when the process actually stops. To observe the actual process stopping, see `observe_stop()`.

**stop\_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.

**work\_dir**

A trait for unicode strings.

## BatchSystemLauncher

```
class IPython.parallel.apps.launcher.BatchSystemLauncher(work_dir=u'.',
                                                         config=None,
                                                         **kwargs)
```

Bases: `IPython.parallel.apps.launcher.BaseLauncher`

Launch an external process using a batch system.

This class is designed to work with UNIX batch systems like PBS, LSF, GridEngine, etc. The overall model is that there are different commands like `qsub`, `qdel`, etc. that handle the starting and stopping of the process.

This class also has the notion of a batch script. The `batch_template` attribute can be set to a string that is a template for the batch script. This template is instantiated using `Itpl`. Thus the template can use `#{n}` for the number of instances. Subclasses can add additional variables to the template dict.

**`__init__`** (*work\_dir=u'.', config=None, \*\*kwargs*)

**`arg_str`**

The string form of the program arguments.

**`args`**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**`batch_file`**

A casting version of the unicode trait.

**`batch_file_name`**

A casting version of the unicode trait.

**`batch_template`**

A casting version of the unicode trait.

**`batch_template_file`**

A casting version of the unicode trait.

**`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.

**`context`**

An instance of a Python dict.

**`default_template`**

A casting version of the unicode trait.

**`delete_command`**

An instance of a Python list.

**`find_args()`**

**`job_array_regexp`**

A casting version of the unicode trait.

**`job_array_template`**

A casting version of the unicode trait.

**`job_id_regexp`**

A casting version of the unicode trait.

**`log`**

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.

**`logname`**

A casting version of the unicode trait.

**`loop`**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*, *cluster\_dir*)

Start *n* copies of the process using a batch system.

**start\_data**

**stop** ()

**stop\_data**

**submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the work\_dir.

## IPClusterLauncher

```
class IPython.parallel.apps.launcher.IPClusterLauncher (work_dir=u'.', con-
                                                         fig=None, **kwargs)
```

Bases: IPython.parallel.apps.launcher.LocalProcessLauncher

Launch the ipcluster program in an external process.

```
__init__ (work_dir=u'.', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args()****handle\_stderr** (*fd, events*)**handle\_stdout** (*fd, events*)**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**ipcluster\_args**

An instance of a Python list.

**ipcluster\_cmd**

An instance of a Python list.

**ipcluster\_n**

A integer trait.

**ipcluster\_subcommand**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()

**poll\_frequency**

A integer trait.

**running**

Am I running.

**signal** (*sig*)

**start** ()

**start\_data**

**stop** ()

**stop\_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.

#### **work\_dir**

A trait for unicode strings.

### **LauncherError**

```
class IPython.parallel.apps.launcher.LauncherError
```

Bases: `exceptions.Exception`

**\_\_init\_\_()**

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

**args**

**message**

### **LocalControllerLauncher**

```
class IPython.parallel.apps.launcher.LocalControllerLauncher (work_dir=u'.',
                                                             con-
                                                             fig=None,
                                                             **kwargs)
```

Bases: `IPython.parallel.apps.launcher.LocalProcessLauncher`

Launch a controller as a regular external process.

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of `cmd` and `args` that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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.

**controller\_args**

An instance of a Python list.

**controller\_cmd**

An instance of a Python list.

**find\_args()**

**handle\_stderr** (*fd, events*)

**handle\_stdout** (*fd, events*)

**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.



**poll()**

**poll\_frequency**  
A integer trait.

**running**  
Am I running.

**signal(sig)**

**start(cluster\_dir)**  
Start the controller by cluster\_dir.

**start\_data**

**stop()**

**stop\_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.

**work\_dir**  
A trait for unicode strings.

### LocalEngineLauncher

```
class IPython.parallel.apps.launcher.LocalEngineLauncher(work_dir=u'.',
                                                         config=None,
                                                         **kwargs)
```

Bases: `IPython.parallel.apps.launcher.LocalProcessLauncher`

Launch a single engine as a regular external process.

**\_\_init\_\_**(work\_dir=u'.', config=None, \*\*kwargs)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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.

**engine\_args**

An instance of a Python list.

**engine\_cmd**

An instance of a Python list.

**find\_args()****handle\_stderr** (*fd, events*)**handle\_stdout** (*fd, events*)**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll()**

**poll\_frequency**

A integer trait.

**running**

Am I running.

**signal** (*sig*)

**start** (*cluster\_dir*)

Start the engine by cluster\_dir.

**start\_data**

**stop()**

**stop\_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.

**work\_dir**

A trait for unicode strings.

**LocalEngineSetLauncher**

```
class IPython.parallel.apps.launcher.LocalEngineSetLauncher (work_dir=u'.',  
                                                             config=None,  
                                                             **kwargs)
```

Bases: `IPython.parallel.apps.launcher.BaseLauncher`

Launch a set of engines as regular external processes.

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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.

**engine\_args**

An instance of a Python list.

**find\_args()**

**interrupt\_then\_kill** (*delay=1.0*)

**launcher\_class**

alias of `LocalEngineLauncher`

**launchers**

An instance of a Python dict.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**running**

Am I running.

**signal** (*sig*)**start** (*n, cluster\_dir*)

Start n engines by profile or cluster\_dir.

**start\_data****stop** ()**stop\_data**

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.

**work\_dir**

A trait for unicode strings.

**LocalProcessLauncher**

```
class IPython.parallel.apps.launcher.LocalProcessLauncher (work_dir=u'.',
                                                            config=None,
                                                            **kwargs)
```

Bases: `IPython.parallel.apps.launcher.BaseLauncher`

Start and stop an external process in an asynchronous manner.

This will launch the external process with a working directory of `self.work_dir`.

```
__init__ (work_dir=u'.', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args()****handle\_stderr** (*fd, events*)**handle\_stdout** (*fd, events*)**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()

**poll\_frequency**

A integer trait.

**running**

Am I running.

**signal** (*sig*)

**start** ()

**start\_data**

**stop** ()

**stop\_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.

**work\_dir**

A trait for unicode strings.

### **MPIExecControllerLauncher**

```
class IPython.parallel.apps.launcher.MPIExecControllerLauncher (work_dir=u'.',
                                                                con-
                                                                fig=None,
                                                                **kwargs)
```

Bases: `IPython.parallel.apps.launcher.MPIExecLauncher`

Launch a controller using mpiexec.

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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.

**controller\_args**

An instance of a Python list.

**controller\_cmd**

An instance of a Python list.

**find\_args** ()

**handle\_stderr** (*fd, events*)



**handle\_stdout** (*fd, events*)

**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**mpi\_args**

An instance of a Python list.

**mpi\_cmd**

An instance of a Python list.

**n**

A integer trait.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll()**

**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)

**start** (*cluster\_dir*)

Start the controller by cluster\_dir.

**start\_data**

**stop()**

**stop\_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.

**work\_dir**

A trait for unicode strings.

**MPIExecEngineSetLauncher**

```
class IPython.parallel.apps.launcher.MPIExecEngineSetLauncher (work_dir=u'.',
                                                                con-
                                                                fig=None,
                                                                **kwargs)
```

Bases: `IPython.parallel.apps.launcher.MPIExecLauncher`

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args()**

Build self.args using all the fields.

**handle\_stderr** (*fd, events*)**handle\_stdout** (*fd, events*)**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**mpi\_args**

An instance of a Python list.

**mpi\_cmd**

An instance of a Python list.

**n**

A integer trait.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()

**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)

**start** (*n, cluster\_dir*)

Start n engines by profile or cluster\_dir.

**start\_data****stop()****stop\_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.

**work\_dir**

A trait for unicode strings.

## **MPIDebugLauncher**

**class** IPython.parallel.apps.launcher.**MPIDebugLauncher** (*work\_dir=u'.', config=None, fig=None, \*\*kwargs*)

Bases: IPython.parallel.apps.launcher.LocalProcessLauncher

Launch an external process using mpiexec.

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args()**

Build self.args using all the fields.

**handle\_stderr** (*fd, events*)**handle\_stdout** (*fd, events*)

**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**mpi\_args**

An instance of a Python list.

**mpi\_cmd**

An instance of a Python list.

**n**

A integer trait.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()

**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)

**start** (*n*)

Start *n* instances of the program using mpiexec.

**start\_data**

**stop** ()

**stop\_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.

**work\_dir**

A trait for unicode strings.

**PBSControllerLauncher**

```
class IPython.parallel.apps.launcher.PBSControllerLauncher (work_dir=u'',
                                                            config=None,
                                                            **kwargs)
```

Bases: `IPython.parallel.apps.launcher.PBSLauncher`

Launch a controller using PBS.

**\_\_init\_\_** (*work\_dir=u'', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.



**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*cluster\_dir*)

Start the controller by profile or cluster\_dir.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the work\_dir.

**PBSEngineSetLauncher**

```
class IPython.parallel.apps.launcher.PBSEngineSetLauncher (work_dir=u'',
                                                           config=None,
                                                           **kwargs)
```

Bases: `IPython.parallel.apps.launcher.PBSLauncher`

Launch Engines using PBS

```
__init__ (work_dir=u'', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** *sig* : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*, *cluster\_dir*)

Start *n* engines by profile or *cluster\_dir*.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the `work_dir`.

**PBSLauncher**

```
class IPython.parallel.apps.launcher.PBSLauncher (work_dir=u'.', config=None,
                                                    **kwargs)
```

Bases: `IPython.parallel.apps.launcher.BatchSystemLauncher`

A BatchSystemLauncher subclass for PBS.

```
__init__ (work_dir=u'.', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*, *cluster\_dir*)

Start *n* copies of the process using a batch system.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the work\_dir.

**ProcessStateError**

**class** IPython.parallel.apps.launcher.**ProcessStateError**

Bases: IPython.parallel.apps.launcher.LauncherError

**\_\_init\_\_** ()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature



**args**

**message**

### **SGEControllerLauncher**

```
class IPython.parallel.apps.launcher.SGEControllerLauncher (work_dir=u'',  
                                                            config=None,  
                                                            **kwargs)
```

Bases: `IPython.parallel.apps.launcher.SGELauncher`

Launch a controller using SGE.

```
__init__(work_dir=u'', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**parse\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*cluster\_dir*)

Start the controller by profile or cluster\_dir.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the work\_dir.

**SGEEngineSetLauncher**

```
class IPython.parallel.apps.launcher.SGEEngineSetLauncher (work_dir=u'',
                                                            config=None,
                                                            **kwargs)
```

Bases: `IPython.parallel.apps.launcher.SGELauncher`

Launch Engines with SGE

**\_\_init\_\_** (*work\_dir=u'', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.

**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*, *cluster\_dir*)

Start *n* engines by profile or *cluster\_dir*.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the *work\_dir*.

**SGELauncher**

**class** IPython.parallel.apps.launcher.**SGELauncher** (*work\_dir=u'.'*, *config=None*,  
*\*\*kwargs*)

Bases: IPython.parallel.apps.launcher.PBSLauncher

Sun GridEngine is a PBS clone with slightly different syntax

**\_\_init\_\_** (*work\_dir=u'.'*, *config=None*, *\*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**batch\_file**

A casting version of the unicode trait.

**batch\_file\_name**

A casting version of the unicode trait.

**batch\_template**

A casting version of the unicode trait.

**batch\_template\_file**

A casting version of the unicode trait.

**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.

**context**

An instance of a Python dict.

**default\_template**

A casting version of the unicode trait.

**delete\_command**

An instance of a Python list.

**find\_args()**

**job\_array\_regexp**

A casting version of the unicode trait.

**job\_array\_template**

A casting version of the unicode trait.

**job\_id\_regexp**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**queue**

A casting version of the unicode trait.

**queue\_regexp**

A casting version of the unicode trait.



**queue\_template**

A casting version of the unicode trait.

**running**

Am I running.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*, *cluster\_dir*)

Start *n* copies of the process using a batch system.

**start\_data****stop** ()**stop\_data****submit\_command**

An instance of a Python list.

**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.

**work\_dir**

A trait for unicode strings.

**write\_batch\_script** (*n*)

Instantiate and write the batch script to the work\_dir.

**SSHControllerLauncher**

```
class IPython.parallel.apps.launcher.SSHControllerLauncher (work_dir=u'.',  
                                                             config=None,  
                                                             **kwargs)
```

Bases: `IPython.parallel.apps.launcher.SSHLauncher`

**\_\_init\_\_** (*work\_dir=u'.'*, *config=None*, *\*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args()**

**handle\_stderr** (*fd, events*)

**handle\_stdout** (*fd, events*)

**hostname**

A casting version of the unicode trait.

**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**location**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)**ssh\_args**

An instance of a Python list.

**ssh\_cmd**

An instance of a Python list.

**start** (*cluster\_dir, hostname=None, user=None*)**start\_data****stop** ()**stop\_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.

**user**

A casting version of the unicode trait.

**work\_dir**

A trait for unicode strings.

## SSHEngineLauncher

**class** IPython.parallel.apps.launcher.SSHEngineLauncher (*work\_dir=u'.', config=None, \*\*kwargs*)

Bases: IPython.parallel.apps.launcher.SSHLauncher

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**cmd\_and\_args**

An instance of a Python list.

**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\_args** ()

**handle\_stderr** (*fd, events*)

**handle\_stdout** (*fd, events*)

**hostname**

A casting version of the unicode trait.

**interrupt\_then\_kill** (*delay=2.0*)

Send INT, wait a delay and then send KILL.

**location**

A casting version of the unicode trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll()**

**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)

**ssh\_args**

An instance of a Python list.

**ssh\_cmd**

An instance of a Python list.

**start** (*cluster\_dir*, *hostname=None*, *user=None*)

**start\_data**

**stop()**

**stop\_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.

**user**

A casting version of the unicode trait.

**work\_dir**

A trait for unicode strings.

**SSHEngineSetLauncher**

```
class IPython.parallel.apps.launcher.SSHEngineSetLauncher (work_dir=u'.',
                                                         config=None,
                                                         **kwargs)
```

Bases: `IPython.parallel.apps.launcher.LocalEngineSetLauncher`

```
__init__ (work_dir=u'.', config=None, **kwargs)
```

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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.

**engine\_args**

An instance of a Python list.

**engines**

An instance of a Python dict.

**find\_args()****interrupt\_then\_kill** (*delay=1.0*)**launcher\_class**

alias of `SSHEngineLauncher`

**launchers**

An instance of a Python dict.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**running**

Am I running.

**signal** (*sig*)

**start** (*n, cluster\_dir*)

Start engines by profile or `cluster_dir`. *n* is ignored, and the `engines` config property is used instead.

**start\_data**

**stop** ()

**stop\_data**

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.

#### **work\_dir**

A trait for unicode strings.

### **SSHLauncher**

```
class IPython.parallel.apps.launcher.SSHLauncher (work_dir=u'.', config=None,
                                                    **kwargs)
```

Bases: `IPython.parallel.apps.launcher.LocalProcessLauncher`

A minimal launcher for ssh.

To be useful this will probably have to be extended to use the `sshx` idea for environment variables. There could be other things this needs as well.

```
__init__ (work_dir=u'.', config=None, **kwargs)
```

#### **arg\_str**

The string form of the program arguments.

#### **args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

#### **cmd\_and\_args**

An instance of a Python list.

#### **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\_args()**

```
handle_stderr (fd, events)
```

```
handle_stdout (fd, events)
```

#### **hostname**

A casting version of the unicode trait.

```
interrupt_then_kill (delay=2.0)
```

Send INT, wait a delay and then send KILL.

#### **location**

A casting version of the unicode trait.

#### **log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**poll** ()**poll\_frequency**

A integer trait.

**program**

An instance of a Python list.

**program\_args**

An instance of a Python list.

**running**

Am I running.

**signal** (*sig*)**ssh\_args**

An instance of a Python list.

**ssh\_cmd**

An instance of a Python list.

**start** (*cluster\_dir*, *hostname=None*, *user=None*)**start\_data****stop** ()**stop\_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.

**user**

A casting version of the unicode trait.

**work\_dir**

A trait for unicode strings.

**UnknownStatus****class** IPython.parallel.apps.launcher.UnknownStatus

Bases: IPython.parallel.apps.launcher.LauncherError

**\_\_init\_\_** ()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args****message**

**WindowsHPCControllerLauncher**

```
class IPython.parallel.apps.launcher.WindowsHPCControllerLauncher (work_dir=u'.',
                                                                    con-
                                                                    fig=None,
                                                                    **kwargs)
```

Bases: `IPython.parallel.apps.launcher.WindowsHPCLauncher`

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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.

**extra\_args**

An instance of a Python list.

**find\_args** ()

**job\_cmd**

A casting version of the unicode trait.

**job\_file**

**job\_file\_name**

A casting version of the unicode trait.

**job\_id\_regexp**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**running**

Am I running.

**scheduler**

A casting version of the unicode trait.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

'KILL', 'INT', etc., or any signal number

**start** (*cluster\_dir*)

Start the controller by `cluster_dir`.

**start\_data****stop** ()

**stop\_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.

**work\_dir**

A trait for unicode strings.

**write\_job\_file** (*n*)

## WindowsHPCEngineSetLauncher

```
class IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher (work_dir=u'.',  
                                                                    con-  
                                                                    fig=None,  
                                                                    **kwargs)
```

Bases: `IPython.parallel.apps.launcher.WindowsHPCLauncher`

**\_\_init\_\_** (*work\_dir=u'.', config=None, \*\*kwargs*)

**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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.

**extra\_args**

An instance of a Python list.

**find\_args** ()

**job\_cmd**

A casting version of the unicode trait.

**job\_file**

**job\_file\_name**

A casting version of the unicode trait.

**job\_id\_regexp**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start** (*data*)

Call this to trigger startup actions.

This logs the process startup and sets the state to ‘running’. It is a pass-through so it can be used as a callback.

**notify\_stop** (*data*)

Call this to trigger process stop actions.

This logs the process stopping and sets the state to ‘after’. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop** (*f*)

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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.

**parse\_job\_id**(*output*)

Take the output of the submit command and return the job id.

**running**

Am I running.

**scheduler**

A casting version of the unicode trait.

**signal**(*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** *sig* : str or int

'KILL', 'INT', etc., or any signal number

**start**(*n*, *cluster\_dir*)

Start the controller by *cluster\_dir*.

**start\_data**

**stop**()

**stop\_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.

**work\_dir**

A trait for unicode strings.

**write\_job\_file**(*n*)

## WindowsHPCLauncher

```
class IPython.parallel.apps.launcher.WindowsHPCLauncher(work_dir=u'.',  
                                                         config=None,  
                                                         **kwargs)
```

Bases: `IPython.parallel.apps.launcher.BaseLauncher`

```
__init__(work_dir=u'.', config=None, **kwargs)
```



**arg\_str**

The string form of the program arguments.

**args**

A list of cmd and args that will be used to start the process.

This is what is passed to `spawnProcess()` and the first element will be the process name.

**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\_args()****job\_cmd**

A casting version of the unicode trait.

**job\_file****job\_file\_name**

A casting version of the unicode trait.

**job\_id\_regexp**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**notify\_start(data)**

Call this to trigger startup actions.

This logs the process startup and sets the state to 'running'. It is a pass-through so it can be used as a callback.

**notify\_stop(data)**

Call this to trigger process stop actions.

This logs the process stopping and sets the state to 'after'. Call this to trigger all the deferreds from `observe_stop()`.

**on\_stop(f)**

Get a deferred that will fire when the process stops.

The deferred will fire with data that contains information about the exit status of the process.

**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\_job\_id** (*output*)

Take the output of the submit command and return the job id.

**running**

Am I running.

**scheduler**

A casting version of the unicode trait.

**signal** (*sig*)

Signal the process.

Return a semi-meaningless deferred after signaling the process.

**Parameters** **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

**start** (*n*)

Start *n* copies of the process using the Win HPC job scheduler.

**start\_data**

**stop** ()

**stop\_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.

**work\_dir**

A trait for unicode strings.

**write\_job\_file**(*n*)

### 8.54.3 Function

`IPython.parallel.apps.launcher.find_job_cmd()`

## 8.55 parallel.apps.logwatcher

### 8.55.1 Module: `parallel.apps.logwatcher`

Inheritance diagram for `IPython.parallel.apps.logwatcher`:



A simple logger object that consolidates messages incoming from ipcluster processes.

### 8.55.2 LogWatcher

**class** `IPython.parallel.apps.logwatcher.LogWatcher` (*\*\*kwargs*)

Bases: `IPython.parallel.factory.LoggingFactory`

A simple class that receives messages on a SUB socket, as published by subclasses of `zmq.log.handlers.PUBHandler`, and logs them itself.

This can subscribe to multiple topics, but defaults to all topics.

**\_\_init\_\_** (*\*\*kwargs*)

**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.

**context**

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.

**log**

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.

**log\_message** (*raw*)

receive and parse a message, then log it.

**logname**

A casting version of the unicode trait.

**loop**

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.

**start** ()**stop** ()**stream**

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.

**subscribe** ()

Update our SUB socket’s subscriptions.

**topics**

An instance of a Python list.

**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.

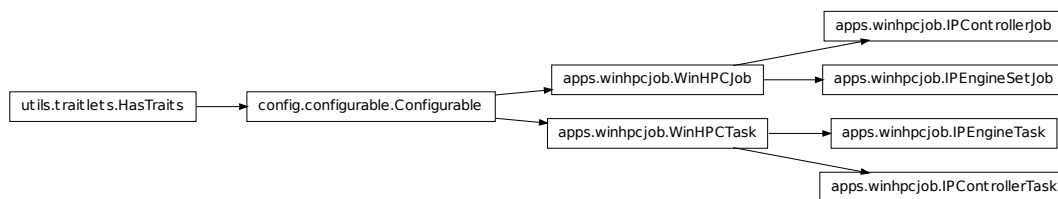
**url**

A trait for strings.

## 8.56 parallel.apps.winhpcjob

### 8.56.1 Module: parallel.apps.winhpcjob

Inheritance diagram for IPython.parallel.apps.winhpcjob:



Job and task components for writing .xml files that the Windows HPC Server 2008 can use to start jobs.

### 8.56.2 Classes

#### IPControllerJob

**class** IPython.parallel.apps.winhpcjob.**IPControllerJob**(\*\**kwargs*)

Bases: IPython.parallel.apps.winhpcjob.WinHPCJob

**\_\_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.

**add\_task** (*task*)

Add a task to the job.

**Parameters** **task**: WinHPCTask

The task object to add.

**as\_element** ()

**auto\_calculate\_max**

A boolean (True, False) trait.

**auto\_calculate\_min**

A boolean (True, False) trait.

**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.

**is\_exclusive**

A boolean (True, False) trait.

**job\_id**

A trait for strings.

**job\_name**

A trait for strings.

**job\_type**

A trait for strings.

**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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.

**owner****priority**

An enum that whose value must be in a given sequence.

**project**

A trait for strings.

**requested\_nodes**

A trait for strings.

**run\_until\_canceled**

A boolean (True, False) trait.

**tasks**

An instance of a Python list.

**tostring** ()

Return the string representation of the job description XML.

**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.

**unit\_type**

A trait for strings.

**username**

A trait for strings.

**version**

A trait for strings.

**write** (*filename*)

Write the XML job description to a file.

**xmlns**

A trait for strings.

**IPControllerTask**

```
class IPython.parallel.apps.winhpcjob.IPControllerTask (config=None)
```

```
    Bases: IPython.parallel.apps.winhpcjob.WinHPCTask
```

```
    __init__ (config=None)
```

```
    as_element ()
```

```
    command_line
```

```
    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.

```
    controller_args
```

An instance of a Python list.

```
    controller_cmd
```

An instance of a Python list.

```
    environment_variables
```

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_env_vars ()
```

```
    is_parametric
```

A boolean (True, False) trait.

```
    is_rerunnable
```

A boolean (True, False) trait.



**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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.

**std\_err\_file\_path**

A casting version of the string trait.

**std\_out\_file\_path**

A casting version of the string trait.

**task\_id**

A trait for strings.

**task\_name**

A trait for strings.

**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.

**unit\_type**

A trait for strings.

**version**

A trait for strings.

**work\_directory**

A casting version of the string trait.

## IPythonSetJob

**class** IPython.parallel.apps.winhpcjob.**IPythonSetJob** (\*\**kwargs*)

Bases: IPython.parallel.apps.winhpcjob.WinHPCJob

**\_\_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.

**add\_task** (*task*)

Add a task to the job.

**Parameters** **task** : WinHPCTask

The task object to add.

**as\_element** ()

**auto\_calculate\_max**

A boolean (True, False) trait.

**auto\_calculate\_min**

A boolean (True, False) trait.

**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.

**is\_exclusive**

A boolean (True, False) trait.

**job\_id**

A trait for strings.

**job\_name**

A trait for strings.

**job\_type**

A trait for strings.

**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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.

**owner**

**priority**

An enum that whose value must be in a given sequence.

**project**

A trait for strings.

**requested\_nodes**

A trait for strings.

**run\_until\_canceled**

A boolean (True, False) trait.

**tasks**

An instance of a Python list.

**tostring** ()

Return the string representation of the job description XML.

**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.

**unit\_type**

A trait for strings.

**username**

A trait for strings.

**version**

A trait for strings.

**write** (*filename*)

Write the XML job description to a file.

**xmlns**

A trait for strings.

**IPEngineTask**

**class** IPython.parallel.apps.winhpcjob.**IPEngineTask** (*config=None*)

Bases: IPython.parallel.apps.winhpcjob.WinHPCTask

**\_\_init\_\_** (*config=None*)

**as\_element** ()

**command\_line**

**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.

**engine\_args**

An instance of a Python list.

**engine\_cmd**

An instance of a Python list.

**environment\_variables**

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\_env\_vars** ()

**is\_parametric**

A boolean (True, False) trait.

**is\_rerunnable**

A boolean (True, False) trait.

**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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 uninstalls it.

**std\_err\_file\_path**

A casting version of the string trait.

**std\_out\_file\_path**

A casting version of the string trait.

**task\_id**

A trait for strings.

**task\_name**

A trait for strings.

**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.

**unit\_type**

A trait for strings.

**version**

A trait for strings.

**work\_directory**

A casting version of the string trait.

**WinHPCJob**

**class** IPython.parallel.apps.winhpcjob.**WinHPCJob** (\*\*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.

**add\_task** (task)

Add a task to the job.

**Parameters** **task** : WinHPCTask

The task object to add.

**as\_element** ()

**auto\_calculate\_max**

A boolean (True, False) trait.

**auto\_calculate\_min**

A boolean (True, False) trait.

**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.

**is\_exclusive**

A boolean (True, False) trait.

**job\_id**

A trait for strings.

**job\_name**

A trait for strings.

**job\_type**

A trait for strings.

**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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 uninstalls it.

**owner****priority**

An enum that whose value must be in a given sequence.

**project**

A trait for strings.

**requested\_nodes**

A trait for strings.

**run\_until\_canceled**

A boolean (True, False) trait.



**tasks**

An instance of a Python list.

**tostring()**

Return the string representation of the job description XML.

**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.

**unit\_type**

A trait for strings.

**username**

A trait for strings.

**version**

A trait for strings.

**write** (*filename*)

Write the XML job description to a file.

**xmlns**

A trait for strings.

**WinHPCTask**

```
class IPython.parallel.apps.winhpcjob.WinHPCTask (**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.

**as\_element()**

**command\_line**

A casting version of the string trait.

**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.

**environment\_variables**

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\_env\_vars()**

**is\_parametric**

A boolean (True, False) trait.

**is\_rerunnable**

A boolean (True, False) trait.

**max\_cores**

A integer trait.

**max\_nodes**

A integer trait.

**max\_sockets**

A integer trait.

**min\_cores**

A integer trait.

**min\_nodes**

A integer trait.

**min\_sockets**

A integer 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.

**std\_err\_file\_path**

A casting version of the string trait.

**std\_out\_file\_path**

A casting version of the string trait.

**task\_id**

A trait for strings.

**task\_name**

A trait for strings.

**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.

**unit\_type**

A trait for strings.

**version**

A trait for strings.

**work\_directory**

A casting version of the string trait.

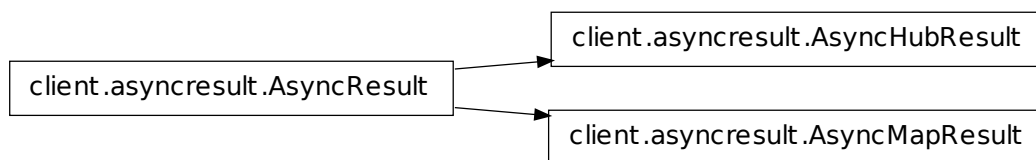
### 8.56.3 Functions

```
IPython.parallel.apps.winhpcjob.as_str(value)
IPython.parallel.apps.winhpcjob.find_username()
IPython.parallel.apps.winhpcjob.indent(elem, level=0)
```

## 8.57 parallel.client.asyncresult

### 8.57.1 Module: parallel.client.asyncresult

Inheritance diagram for IPython.parallel.client.asyncresult:



AsyncResult objects for the client

### 8.57.2 Classes

#### AsyncHubResult

```
class IPython.parallel.client.asyncresult.AsyncHubResult(client, msg_ids,
                                                         fname='unknown',
                                                         targets=None,
                                                         tracker=None)
```

Bases: IPython.parallel.client.asyncresult.AsyncResult

Class to wrap pending results that must be requested from the Hub.

Note that waiting/polling on these objects requires polling the Hub over the network, so use `AsyncHubResult.wait()` sparingly.

```
__init__(client, msg_ids, fname='unknown', targets=None, tracker=None)
```

```
abort()
    abort my tasks.
```

```
get(timeout=-1)
    Return the result when it arrives.
```

If *timeout* is not None and the result does not arrive within *timeout* seconds then `TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()` inside a *RemoteError*.

**get\_dict** (*timeout=-1*)

Get the results as a dict, keyed by `engine_id`.

*timeout* behavior is described in `get()`.

**metadata**

property for accessing execution metadata.

**r**

result property wrapper for `get(timeout=0)`.

**ready** ()

Return whether the call has completed.

**result**

result property wrapper for `get(timeout=0)`.

**result\_dict**

result property as a dict.

**sent**

check whether my messages have been sent.

**successful** ()

Return whether the call completed without raising an exception.

Will raise `AssertionError` if the result is not ready.

**wait** (*timeout=-1*)

wait for result to complete.

**wait\_for\_send** (*timeout=-1*)

wait for pyzmq send to complete.

This is necessary when sending arrays that you intend to edit in-place. *timeout* is in seconds, and will raise `TimeoutError` if it is reached before the send completes.

## AsyncMapResult

```
class IPython.parallel.client.asyncresult.AsyncMapResult (client, msg_ids,
                                                         mapObject,
                                                         fname='')
    Bases: IPython.parallel.client.asyncresult.AsyncResult
```

Class for representing results of non-blocking gathers.

This will properly reconstruct the gather.

```
__init__ (client, msg_ids, mapObject, fname='')
```

**abort** ()

abort my tasks.

**get** (*timeout=-1*)

Return the result when it arrives.

If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()` inside a `RemoteError`.

**get\_dict** (*timeout=-1*)

Get the results as a dict, keyed by `engine_id`.

*timeout* behavior is described in `get()`.

**metadata**

property for accessing execution metadata.

**r**

result property wrapper for `get(timeout=0)`.

**ready** ()

Return whether the call has completed.

**result**

result property wrapper for `get(timeout=0)`.

**result\_dict**

result property as a dict.

**sent**

check whether my messages have been sent.

**successful** ()

Return whether the call completed without raising an exception.

Will raise `AssertionError` if the result is not ready.

**wait** (*timeout=-1*)

Wait until the result is available or until *timeout* seconds pass.

This method always returns `None`.

**wait\_for\_send** (*timeout=-1*)

wait for pyzmq send to complete.

This is necessary when sending arrays that you intend to edit in-place. *timeout* is in seconds, and will raise `TimeoutError` if it is reached before the send completes.

## AsyncResult

```
class IPython.parallel.client.asyncresult.AsyncResult (client, msg_ids,
                                                         fname='unknown',
                                                         targets=None,
                                                         tracker=None)
```

Bases: `object`

Class for representing results of non-blocking calls.

Provides the same interface as `multiprocessing.pool.AsyncResult`.

**\_\_init\_\_** (*client, msg\_ids, fname='unknown', targets=None, tracker=None*)

**abort** ()

abort my tasks.

**get** (*timeout=-1*)

Return the result when it arrives.

If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()` inside a `RemoteError`.

**get\_dict** (*timeout=-1*)

Get the results as a dict, keyed by `engine_id`.

*timeout* behavior is described in `get()`.

**metadata**

property for accessing execution metadata.

**r**

result property wrapper for `get(timeout=0)`.

**ready** ()

Return whether the call has completed.

**result**

result property wrapper for `get(timeout=0)`.

**result\_dict**

result property as a dict.

**sent**

check whether my messages have been sent.

**successful** ()

Return whether the call completed without raising an exception.

Will raise `AssertionError` if the result is not ready.

**wait** (*timeout=-1*)

Wait until the result is available or until *timeout* seconds pass.

This method always returns `None`.

**wait\_for\_send** (*timeout=-1*)

wait for `pzmq` send to complete.

This is necessary when sending arrays that you intend to edit in-place. *timeout* is in seconds, and will raise `TimeoutError` if it is reached before the send completes.

### 8.57.3 Function

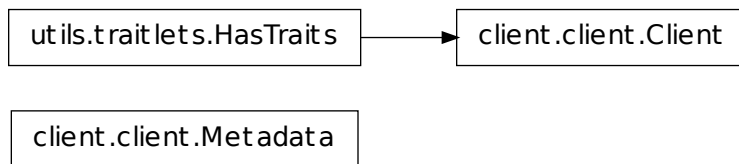
`IPython.parallel.client.asyncresult.check_ready(f)`

Call `spin()` to sync state prior to calling the method.

## 8.58 `parallel.client.client`

### 8.58.1 Module: `parallel.client.client`

Inheritance diagram for `IPython.parallel.client.client`:



A semi-synchronous Client for the ZMQ cluster

### 8.58.2 Classes

#### `Client`

```
class IPython.parallel.client.client.Client(url_or_file=None, profile='default',
                                             cluster_dir=None, ipython_dir=None,
                                             context=None, username=None,
                                             debug=False, exec_key=None,
                                             sshserver=None, sshkey=None,
                                             password=None, paramiko=None,
                                             timeout=10)
```

Bases: `IPython.utils.traits.HasTraits`

A semi-synchronous client to the IPython ZMQ cluster

**Parameters** `url_or_file` : bytes; zmq url or path to ipcontroller-client.json

Connection information for the Hub's registration. If a json connector file is given, then likely no further configuration is necessary. [Default: use profile]

**profile** : bytes

The name of the Cluster profile to be used to find connector information. [Default: 'default']



**context** : zmq.Context

Pass an existing zmq.Context instance, otherwise the client will create its own.

**username** : bytes

set username to be passed to the Session object

**debug** : bool

flag for lots of message printing for debug purposes

**#———— ssh related args ———— :**

**# These are args for configuring the ssh tunnel to be used :**

**# credentials are used to forward connections over ssh to the Controller :**

**# Note that the ip given in ‘addr‘ needs to be relative to sshserver :**

**# The most basic case is to leave addr as pointing to localhost (127.0.0.1), :**

**# and set sshserver as the same machine the Controller is on. However, :**

**# the only requirement is that sshserver is able to see the Controller :**

**# (i.e. is within the same trusted network). :**

**sshserver** : str

A string of the form passed to ssh, i.e. ‘server.tld’ or ‘user@server.tld:port’ If keyfile or password is specified, and this is not, it will default to the ip given in addr.

**sshkey** : str; path to public ssh key file

This specifies a key to be used in ssh login, default None. Regular default ssh keys will be used without specifying this argument.

**password** : str

Your ssh password to sshserver. Note that if this is left None, you will be prompted for it if passwordless key based login is unavailable.

**paramiko** : bool

flag for whether to use paramiko instead of shell ssh for tunneling. [default: True on win32, False else]

**———— exec authentication args ———— :**

**If even localhost is untrusted, you can have some protection against :**

**unauthorized execution by using a key. Messages are still sent :**

**as cleartext, so if someone can snoop your loopback traffic this will :**

**not help against malicious attacks. :**

**exec\_key** : str

an authentication key or file containing a key default: None

**Attributes** **ids** : list of int engine IDs

requesting the ids attribute always synchronizes the registration state. To request ids without synchronization, use semi-private `_ids` attributes.

**history** : list of msg\_ids

a list of msg\_ids, keeping track of all the execution messages you have submitted in order.

**outstanding** : set of msg\_ids

a set of msg\_ids that have been submitted, but whose results have not yet been received.

**results** : dict

a dict of all our results, keyed by msg\_id

**block** : bool

determines default behavior when block not specified in execution methods

**Methods** **spin** :

flushes incoming results and registration state changes control methods spin, and requesting *ids* also ensures up to date

**wait** :

wait on one or more msg\_ids

**execution methods** :

apply legacy: execute, run

**data movement** :

push, pull, scatter, gather

**query methods** :

queue\_status, get\_result, purge, result\_status

**control methods** :

abort, shutdown

```
__init__(url_or_file=None, profile='default', cluster_dir=None, ipython_dir=None, context=None, username=None, debug=False, exec_key=None, sshserver=None, sshkey=None, password=None, paramiko=None, timeout=10)
```

**abort** (jobs=None, targets=None, block=None)

Abort specific jobs from the execution queues of target(s).

This is a mechanism to prevent jobs that have already been submitted from executing.

**Parameters** **jobs** : msg\_id, list of msg\_ids, or AsyncResult

The jobs to be aborted

**block**

A boolean (True, False) trait.

**clear** (*targets=None, block=None*)

Clear the namespace in target(s).

**close** ()

**debug**

A boolean (True, False) trait.

**direct\_view** (*targets='all'*)

construct a DirectView object.

If no targets are specified, create a DirectView using all engines.

**Parameters** **targets:** list,slice,int,etc. [default: use all engines] :

The engines to use for the View

**get\_result** (*indices\_or\_msg\_ids=None, block=None*)

Retrieve a result by msg\_id or history index, wrapped in an AsyncResult object.

If the client already has the results, no request to the Hub will be made.

This is a convenient way to construct AsyncResult objects, which are wrappers that include metadata about execution, and allow for awaiting results that were not submitted by this Client.

It can also be a convenient way to retrieve the metadata associated with blocking execution, since it always retrieves

**Parameters** **indices\_or\_msg\_ids** : integer history index, str msg\_id, or list of either

The indices or msg\_ids of indices to be retrieved

**block** : bool

Whether to wait for the result to be done

**Returns** **AsyncResult** :

A single AsyncResult object will always be returned.

**AsyncHubResult** :

A subclass of AsyncResult that retrieves results from the Hub

## Examples

```
In [10]: r = client.apply()
```

**history**

An instance of a Python list.

**ids**

Always up-to-date ids property.

**load\_balanced\_view** (*targets=None*)

construct a DirectView object.

If no arguments are specified, create a LoadBalancedView using all engines.

**Parameters** **targets**: list,slice,int,etc. [default: use all engines] :

The subset of engines across which to load-balance

**metadata**

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.

**outstanding**

An instance of a Python set.

**profile**

A casting version of the unicode trait.

**purge\_results** (*jobs=[ ], targets=[ ]*)

Tell the Hub to forget results.

Individual results can be purged by msg\_id, or the entire history of specific targets can be purged.

**Parameters** **jobs** : str or list of str or AsyncResult objects

the msg\_ids whose results should be forgotten.

**targets** : int/str/list of ints/strs

The targets, by uuid or int\_id, whose entire history is to be purged. Use *targets='all'* to scrub everything from the Hub’s memory.

default : None

**queue\_status** (*targets='all', verbose=False*)

Fetch the status of engine queues.

**Parameters** **targets** : int/str/list of ints/strs

the engines whose states are to be queried. default : all

**verbose** : bool

Whether to return lengths only, or lists of ids for each element

**result\_status** (*msg\_ids, status\_only=True*)

Check on the status of the result(s) of the apply request with *msg\_ids*.

If *status\_only* is False, then the actual results will be retrieved, else only the status of the results will be checked.

**Parameters** **msg\_ids** : list of msg\_ids

**if int:** Passed as index to self.history for convenience.

**status\_only** : bool (default: True)

**if False:** Retrieve the actual results of completed tasks.

**Returns** **results** : dict

There will always be the keys 'pending' and 'completed', which will be lists of msg\_ids that are incomplete or complete. If *status\_only* is False, then completed results will be keyed by their *msg\_id*.

**results**

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.

**send\_apply\_message** (*socket, f, args=None, kwargs=None, subheader=None, track=False, ident=None*)

construct and send an apply message via a socket.

This is the principal method with which all engine execution is performed by views.

**shutdown** (*targets=None, restart=False, hub=False, block=None*)

Terminates one or more engine processes, optionally including the hub.

**spin** ()

Flush any registration notifications and execution results waiting in the ZMQ queue.

**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.

**wait** (*jobs=None, timeout=-1*)

waits on one or more *jobs*, for up to *timeout* seconds.

**Parameters** **jobs** : int, str, or list of ints and/or strs, or one or more `AsyncResult` objects

ints are indices to `self.history` strs are `msg_ids` default: wait on all outstanding messages

**timeout** : float

a time in seconds, after which to give up. default is -1, which means no timeout

**Returns** **True** : when all `msg_ids` are done

**False** : timeout reached, some `msg_ids` still outstanding

## Metadata

**class** `IPython.parallel.client.client.Metadata (*args, **kwargs)`

Bases: `dict`

Subclass of `dict` for initializing metadata values.

Attribute access works on keys.

These objects have a strict set of keys - errors will raise if you try to add new keys.

**\_\_init\_\_** (*\*args, \*\*kwargs*)

**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

### 8.58.3 Functions

`IPython.parallel.client.client.default_block(f)`

Default to self.block; preserve self.block.

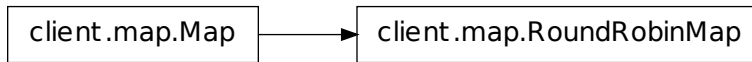
`IPython.parallel.client.client.spin_first(f)`

Call spin() to sync state prior to calling the method.

## 8.59 parallel.client.map

### 8.59.1 Module: parallel.client.map

Inheritance diagram for `IPython.parallel.client.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.

### 8.59.2 Classes

#### Map

**class** `IPython.parallel.client.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.parallel.client.map.RoundRobinMap`  
Bases: `IPython.parallel.client.map.Map`

Partitions a sequence in a round robin fashion.

This currently does not work!

**concatenate** (*listOfPartitions*)

**flatten\_array** (*klass, listOfPartitions*)

**flatten\_list** (*listOfPartitions*)

**getPartition** (*seq, p, q*)

**joinPartitions** (*listOfPartitions*)

### 8.59.3 Function

`IPython.parallel.client.map.mappable` (*obj*)  
return whether an object is mappable or not.



## 8.60 parallel.client.remotefunction

### 8.60.1 Module: `parallel.client.remotefunction`

Inheritance diagram for `IPython.parallel.client.remotefunction`:



Remote Functions and decorators for Views.

### 8.60.2 Classes

#### `ParallelFunction`

```

class IPython.parallel.client.remotefunction.ParallelFunction (view, f,
                                                                dist='b',
                                                                block=None,
                                                                chunk-
                                                                size=None,
                                                                **flags)
  
```

Bases: `IPython.parallel.client.remotefunction.RemoteFunction`

Class for mapping a function to sequences.

This will distribute the sequences according the a mapper, and call the function on each sub-sequence. If called via map, then the function will be called once on each element, rather that each sub-sequence.

**Parameters** **view** : View instance

The view to be used for execution

**f** : callable

The function to be wrapped into a remote function

**dist** : str [default: 'b']

The key for which mapObject to use to distribute sequences options are:

- 'b' : use contiguous chunks in order
- 'r' : use round-robin striping

**block** : bool [default: None]

Whether to wait for results or not. The default behavior is to use the current *block* attribute of *view*

**chunksize** : int or None

The size of chunk to use when breaking up sequences in a load-balanced manner

**\*\*flags** : remaining kwargs are passed to View.temp\_flags

**\_\_init\_\_** (view, f, dist='b', block=None, chunksize=None, \*\*flags)

**map** (\*sequences)

call a function on each element of a sequence remotely. This should behave very much like the builtin map, but return an AsyncMapResult if self.block is False.

### RemoteFunction

```
class IPython.parallel.client.remotefunction.RemoteFunction (view, f,
                                                            block=None,
                                                            **flags)
```

Bases: object

Turn an existing function into a remote function.

**Parameters** **view** : View instance

The view to be used for execution

**f** : callable

The function to be wrapped into a remote function

**block** : bool [default: None]

Whether to wait for results or not. The default behavior is to use the current *block* attribute of *view*

**\*\*flags** : remaining kwargs are passed to View.temp\_flags

**\_\_init\_\_** (view, f, block=None, \*\*flags)

### 8.60.3 Functions

```
IPython.parallel.client.remotefunction.parallel (view, dist='b', block=None,
                                                  **flags)
```

Turn a function into a parallel remote function.

This method can be used for map:

**In [1]: @parallel(view, block=True) ...: def func(a): ...: pass**

```
IPython.parallel.client.remotefunction.remote (view, block=None, **flags)
```

Turn a function into a remote function.

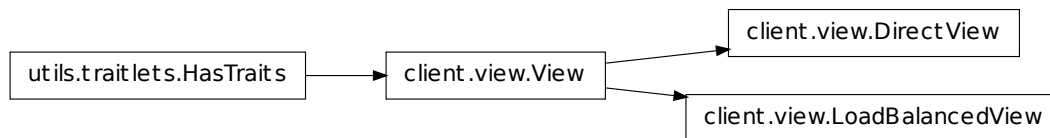
This method can be used for map:

**In [1]: @remote(view, block=True) ...: def func(a): ...: pass**

## 8.61 parallel.client.view

### 8.61.1 Module: parallel.client.view

Inheritance diagram for `IPython.parallel.client.view`:



Views of remote engines.

### 8.61.2 Classes

#### DirectView

**class** `IPython.parallel.client.view.DirectView`(*client=None, socket=None, targets=None*)

Bases: `IPython.parallel.client.view.View`

Direct Multiplexer View of one or more engines.

These are created via indexed access to a client:

```

>>> dv_1 = client[1]
>>> dv_all = client[:]
>>> dv_even = client[::2]
>>> dv_some = client[1:3]
  
```

This object provides dictionary access to engine namespaces:

```
# push a=5: >>> dv['a'] = 5 # pull 'foo': >>> db['foo']
```

**\_\_init\_\_**(*client=None, socket=None, targets=None*)

**abort**(*jobs=None, targets=None, block=None*)

Abort jobs on my engines.

**Parameters** **jobs** : None, str, list of strs, optional

if None: abort all jobs. else: abort specific msg\_id(s).

**activate**( )

Make this *View* 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 *Views* created and used by

the user, there is only one active one. The active *View* is used whenever the magic commands `%px` and `%autopx` are used.

The `activate()` method is called on a given *View* to make it active. Once this has been done, the magic commands can be used.

**apply** (*f*, \**args*, \*\**kwargs*)

calls `f(*args, **kwargs)` on remote engines, returning the result.

This method sets all apply flags via this *View*'s attributes.

**if self.block is False:** returns `AsyncResult`

**else:** returns actual result of `f(*args, **kwargs)`

**apply\_async** (*f*, \**args*, \*\**kwargs*)

calls `f(*args, **kwargs)` on remote engines in a nonblocking manner.

returns `AsyncResult`

**apply\_sync** (*f*, \**args*, \*\**kwargs*)

calls `f(*args, **kwargs)` on remote engines in a blocking manner, returning the result.

returns: actual result of `f(*args, **kwargs)`

**block**

A boolean (True, False) trait.

**clear** (*targets=None*, *block=False*)

Clear the remote namespaces on my engines.

**client**

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.

**execute** (*code*, *targets=None*, *block=None*)

Executes *code* on *targets* in blocking or nonblocking manner.

`execute` is always *bound* (affects engine namespace)

**Parameters** **code** : str

the code string to be executed

**block** : bool

whether or not to wait until done to return default: `self.block`

**gather** (*key*, *dist='b'*, *targets=None*, *block=None*)

Gather a partitioned sequence on a set of engines as a single local seq.

**get** (*key\_s*)

get object(s) by *key\_s* from remote namespace

see *pull* for details.

**get\_result** (*indices\_or\_msg\_ids=None*)

return one or more results, specified by history index or msg\_id.

See `client.get_result` for details.

**history**

An instance of a Python list.

**imap** (*f, \*sequences, \*\*kwargs*)

Parallel version of *itertools.imap*.

See *self.map* for details.

**importer**

`sync_imports(local=True)` as a property.

See `sync_imports` for details.

**kill** (*targets=None, block=True*)

Kill my engines.

**map** (*f, \*sequences, \*\*kwargs*)

`view.map(f, *sequences, block=self.block) => list[AsyncResult]`

Parallel version of builtin *map*, using this View's *targets*.

There will be one task per target, so work will be chunked if the sequences are longer than *targets*.

Results can be iterated as they are ready, but will become available in chunks.

**Parameters** **f**: callable

function to be mapped

**\*sequences: one or more sequences of matching length :**

the sequences to be distributed and passed to *f*

**block** : bool

whether to wait for the result or not [default `self.block`]

**Returns** **if block=False** :

**AsyncResult** An object like `AsyncResult`, but which reassembles the sequence of results into a single list. `AsyncResult`s can be iterated through before all results are complete.

**else** :

**list** the result of `map(f,*sequences)`

**map\_async** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=False)`

See *self.map* for details.

**map\_sync** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=True)`

See *self.map* for details.

**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.

**outstanding**

An instance of a Python set.

**parallel** (*dist='b', block=None, \*\*flags*)

Decorator for making a ParallelFunction

**pull** (*names, targets=None, block=True*)

get object(s) by *name* from remote namespace

will return one object if it is a key. can also take a list of keys, in which case it will return a list of objects.

**purge\_results** (*jobs=[ ], targets=[ ]*)

Instruct the controller to forget specific results.

**push** (*ns, targets=None, block=None, track=None*)

update remote namespace with dict *ns*

**Parameters** **ns** : dict

dict of keys with which to update engine namespace(s)

**block** : bool [default

whether to wait to be notified of engine receipt

**queue\_status** (*targets=None, verbose=False*)

Fetch the Queue status of my engines

**remote** (*block=True, \*\*flags*)

Decorator for making a RemoteFunction

**results**

An instance of a Python dict.

**run** (*filename, targets=None, block=None*)

Execute contents of *filename* on my engine(s).

This simply reads the contents of the file and calls *execute*.

**Parameters** **filename** : str

The path to the file

**targets** : int/str/list of ints/strs

the engines on which to execute default : all

**block** : bool

whether or not to wait until done default: self.block

**scatter** (*key, seq, dist='b', flatten=False, targets=None, block=None, track=None*)

Partition a Python sequence and send the partitions to a set of engines.

**set\_flags** (*\*\*kwargs*)

set my attribute flags by keyword.

Views determine behavior with a few attributes (*block, track*, etc.). These attributes can be set all at once by name with this method.

**Parameters** **block** : bool

whether to wait for results

**track** : bool

whether to create a MessageTracker to allow the user to safely edit after arrays and buffers during non-copying sends.

**shutdown** (*targets=None, restart=False, hub=False, block=None*)

Terminates one or more engine processes, optionally including the hub.

**spin** ()

spin the client, and sync

**sync\_imports** (*\*args, \*\*kws*)

Context Manager for performing simultaneous local and remote imports.

'import x as y' will *not* work. The 'as y' part will simply be ignored.

```
>>> with view.sync_imports():
...     from numpy import recarray
importing recarray from numpy on engine(s)
```

**targets**

**temp\_flags** (\*args, \*\*kws)  
temporarily set flags, for use in *with* statements.  
See `set_flags` for permanent setting of flags

### Examples

```
>>> view.track=False
...
>>> with view.temp_flags(track=True):
...     ar = view.apply(dostuff, my_big_array)
...     ar.tracker.wait() # wait for send to finish
>>> view.track
False
```

**track**  
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 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** (ns)  
update remote namespace with dict *ns*  
See *push* for details.

**wait** (jobs=None, timeout=-1)  
waits on one or more *jobs*, for up to *timeout* seconds.

**Parameters** **jobs** : int, str, or list of ints and/or strs, or one or more AsyncResult objects

ints are indices to self.history strs are msg\_ids default: wait on all outstanding messages

**timeout** : float

a time in seconds, after which to give up. default is -1, which means no timeout

**Returns** **True** : when all msg\_ids are done



**False** : timeout reached, some msg\_ids still outstanding

### LoadBalancedView

**class** IPython.parallel.client.view.**LoadBalancedView** (*client=None*,  
*socket=None, \*\*flags*)

Bases: IPython.parallel.client.view.View

An load-balancing View that only executes via the Task scheduler.

Load-balanced views can be created with the client's *view* method:

```
>>> v = client.load_balanced_view()
```

or targets can be specified, to restrict the potential destinations:

```
>>> v = client.client.load_balanced_view([1, 3])
```

which would restrict loadbalancing to between engines 1 and 3.

**\_\_init\_\_** (*client=None, socket=None, \*\*flags*)

**abort** (*jobs=None, targets=None, block=None*)

Abort jobs on my engines.

**Parameters** **jobs** : None, str, list of str, optional

if None: abort all jobs. else: abort specific msg\_id(s).

**after**

**apply** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines, returning the result.

This method sets all apply flags via this View's attributes.

**if self.block is False:** returns AsyncResult

**else:** returns actual result of *f(\*args, \*\*kwargs)*

**apply\_async** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines in a nonblocking manner.

returns AsyncResult

**apply\_sync** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines in a blocking manner, returning the result.

returns: actual result of *f(\*args, \*\*kwargs)*

**block**

A boolean (True, False) trait.

**client**

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.

**follow**

**get\_result** (*indices\_or\_msg\_ids=None*)

return one or more results, specified by history index or msg\_id.

See `client.get_result` for details.

**history**

An instance of a Python list.

**imap** (*f, \*sequences, \*\*kwargs*)

Parallel version of *itertools.imap*.

See *self.map* for details.

**map** (*f, \*sequences, \*\*kwargs*)

`view.map(f, *sequences, block=self.block, chunksize=1) => list[AsyncResult]`

Parallel version of builtin *map*, load-balanced by this View.

*block*, and *chunksize* can be specified by keyword only.

Each *chunksize* elements will be a separate task, and will be load-balanced. This lets individual elements be available for iteration as soon as they arrive.

**Parameters** *f* : callable

function to be mapped

**\*sequences: one or more sequences of matching length :**

the sequences to be distributed and passed to *f*

**block** : bool

whether to wait for the result or not [default *self.block*]

**track** : bool

whether to create a `MessageTracker` to allow the user to safely edit after arrays and buffers during non-copying sends.

**chunksize** : int

how many elements should be in each task [default 1]

**Returns** if **block=False** :

**AsyncResult** An object like `AsyncResult`, but which reassembles the sequence of results into a single list. `AsyncResult`s can be iterated through before all results are complete.

**else:** the result of `map(f,*sequences)`

**map\_async** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=False)`

See *self.map* for details.

**map\_sync** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=True)`

See *self.map* for details.

**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.

**outstanding**

An instance of a Python set.

**parallel** (*dist='b', block=None, \*\*flags*)

Decorator for making a ParallelFunction

**purge\_results** (*jobs=[], targets=[]*)

Instruct the controller to forget specific results.

**queue\_status** (*targets=None, verbose=False*)

Fetch the Queue status of my engines

**remote** (*block=True, \*\*flags*)

Decorator for making a RemoteFunction

**results**

An instance of a Python dict.

**set\_flags** (*\*\*kwargs*)

set my attribute flags by keyword.

A View is a wrapper for the Client's `apply` method, but with attributes that specify keyword arguments, those attributes can be set by keyword argument with this method.

**Parameters** **block** : bool

whether to wait for results

**track** : bool

whether to create a MessageTracker to allow the user to safely edit after arrays and buffers during non-copying sends.

# :

**after** : Dependency or collection of msg\_ids

Only for load-balanced execution (targets=None) Specify a list of msg\_ids as a time-based dependency. This job will only be run *after* the dependencies have been met.

**follow** : Dependency or collection of msg\_ids

Only for load-balanced execution (targets=None) Specify a list of msg\_ids as a location-based dependency. This job will only be run on an engine where this dependency is met.

**timeout** : float/int or None

Only for load-balanced execution (targets=None) Specify an amount of time (in seconds) for the scheduler to wait for dependencies to be met before failing with a DependencyTimeout.

**shutdown** (targets=None, restart=False, hub=False, block=None)

Terminates one or more engine processes, optionally including the hub.

**spin** ()

spin the client, and sync

**targets**

**temp\_flags** (\*args, \*\*kws)

temporarily set flags, for use in *with* statements.

See set\_flags for permanent setting of flags

## Examples

```
>>> view.track=False
...
>>> with view.temp_flags(track=True):
...     ar = view.apply(dostuff, my_big_array)
...     ar.tracker.wait() # wait for send to finish
>>> view.track
False
```

**timeout**

A casting version of the float trait.

**track**

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 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.

**wait** (*jobs=None, timeout=-1*)

waits on one or more *jobs*, for up to *timeout* seconds.

**Parameters** **jobs** : int, str, or list of ints and/or strs, or one or more AsyncResult objects

ints are indices to self.history strs are msg\_ids default: wait on all outstanding messages

**timeout** : float

a time in seconds, after which to give up. default is -1, which means no timeout

**Returns** **True** : when all msg\_ids are done

**False** : timeout reached, some msg\_ids still outstanding

## View

**class** IPython.parallel.client.view.**View** (*client=None, socket=None, \*\*flags*)

Bases: IPython.utils.traitlets.HasTraits

Base View class for more convenient apply(f,\*args,\*\*kwargs) syntax via attributes.

Don't use this class, use subclasses.

**Methods** **spin** :

flushes incoming results and registration state changes control methods spin, and requesting *ids* also ensures up to date

**wait** :

wait on one or more msg\_ids

**execution methods** :

apply legacy: execute, run

**data movement** :

push, pull, scatter, gather

**query methods :**

get\_result, queue\_status, purge\_results, result\_status

**control methods :**

abort, shutdown

**\_\_init\_\_** (*client=None, socket=None, \*\*flags*)

**abort** (*jobs=None, targets=None, block=None*)

Abort jobs on my engines.

**Parameters jobs** : None, str, list of strs, optional

if None: abort all jobs. else: abort specific msg\_id(s).

**apply** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines, returning the result.

This method sets all apply flags via this View's attributes.

**if self.block is False:** returns AsyncResult

**else:** returns actual result of *f(\*args, \*\*kwargs)*

**apply\_async** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines in a nonblocking manner.

returns AsyncResult

**apply\_sync** (*f, \*args, \*\*kwargs*)

calls *f(\*args, \*\*kwargs)* on remote engines in a blocking manner, returning the result.

returns: actual result of *f(\*args, \*\*kwargs)*

**block**

A boolean (True, False) trait.

**client**

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\_result** (*indices\_or\_msg\_ids=None*)

return one or more results, specified by history index or msg\_id.

See *client.get\_result* for details.

**history**

An instance of a Python list.

**imap** (*f, \*sequences, \*\*kwargs*)

Parallel version of *itertools.imap*.

See *self.map* for details.

**map** (*f, \*sequences, \*\*kwargs*)

override in subclasses

**map\_async** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=False)`

See *self.map* for details.

**map\_sync** (*f, \*sequences, \*\*kwargs*)

Parallel version of builtin *map*, using this view's engines.

This is equivalent to `map(...block=True)`

See *self.map* for details.

**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.

**outstanding**

An instance of a Python set.

**parallel** (*dist='b', block=None, \*\*flags*)

Decorator for making a ParallelFunction

**purge\_results** (*jobs=[ ], targets=[ ]*)

Instruct the controller to forget specific results.

**queue\_status** (*targets=None, verbose=False*)

Fetch the Queue status of my engines

**remote** (*block=True, \*\*flags*)

Decorator for making a RemoteFunction

**results**

An instance of a Python dict.

**set\_flags** (*\*\*kwargs*)

set my attribute flags by keyword.

Views determine behavior with a few attributes (*block*, *track*, etc.). These attributes can be set all at once by name with this method.

**Parameters** **block** : bool

whether to wait for results

**track** : bool

whether to create a `MessageTracker` to allow the user to safely edit after arrays and buffers during non-copying sends.

**shutdown** (*targets=None, restart=False, hub=False, block=None*)

Terminates one or more engine processes, optionally including the hub.

**spin** ()

spin the client, and sync

**targets**

**temp\_flags** (*\*args, \*\*kws*)

temporarily set flags, for use in *with* statements.

See `set_flags` for permanent setting of flags

## Examples

```
>>> view.track=False
...
>>> with view.temp_flags(track=True):
...     ar = view.apply(dostuff, my_big_array)
...     ar.tracker.wait() # wait for send to finish
>>> view.track
False
```

**track**

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 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.



**wait** (*jobs=None, timeout=-1*)

waits on one or more *jobs*, for up to *timeout* seconds.

**Parameters** **jobs** : int, str, or list of ints and/or strs, or one or more AsyncResult objects

ints are indices to self.history strs are msg\_ids default: wait on all outstanding messages

**timeout** : float

a time in seconds, after which to give up. default is -1, which means no timeout

**Returns** **True** : when all msg\_ids are done

**False** : timeout reached, some msg\_ids still outstanding

### 8.61.3 Functions

IPython.parallel.client.view.**save\_ids** (*f*)

Keep our history and outstanding attributes up to date after a method call.

IPython.parallel.client.view.**spin\_after** (*f*)

call spin after the method.

IPython.parallel.client.view.**sync\_results** (*f*)

sync relevant results from self.client to our results attribute.

## 8.62 parallel.controller.controller

### 8.62.1 Module: parallel.controller.controller

Inheritance diagram for IPython.parallel.controller.controller:



The IPython Controller with 0MQ This is a collection of one Hub and several Schedulers.

### 8.62.2 ControllerFactory

**class** IPython.parallel.controller.controller.**ControllerFactory** (*\*\*kwargs*)

Bases: IPython.parallel.controller.hub.HubFactory

Configurable for setting up a Hub and Schedulers.

**`__init__`** (*\*\*kwargs*)

**`children`**

An instance of a Python list.

**`client_ip`**

A casting version of the string trait.

**`client_transport`**

A casting version of the string trait.

**`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.

**`construct`** ()

**`construct_hub`** ()

construct

**`construct_schedulers`** ()

**`context`**

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.

**`control`**

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`**

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_class`**

A casting version of the string trait.

**`engine_ip`**

A casting version of the string trait.

**`engine_transport`**

A casting version of the string trait.

**`exec_key`**

A casting version of the unicode trait.

**`hb`**

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.

**`heartmonitor`**

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.

**hwm**

A integer trait.

**ident**

A casting version of the string trait.

**iopub**

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.

**ip**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**mon\_port**

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.

**monitor\_ip**

A casting version of the string trait.

**monitor\_transport**

A casting version of the string trait.

**monitor\_url**

A casting version of the string trait.

**mq\_class**

A casting version of the string trait.

**mux**

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.

**notifier\_port**

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.

**packer**

A trait for strings.

**ping**

A integer trait.

**regport**

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.

**scheme**

A trait for strings.

**session**

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** ()

**subconstructors**

An instance of a Python list.

**task**

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.

**transport**

A trait for strings.

**unpacker**

A trait for strings.

**url**

A trait for strings.

**username**

A casting version of the unicode trait.

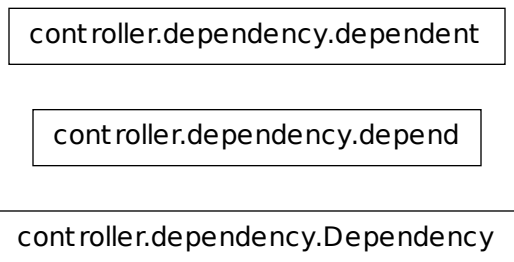
**usethreads**

A boolean (True, False) trait.

## 8.63 parallel.controller.dependency

### 8.63.1 Module: `parallel.controller.dependency`

Inheritance diagram for `IPython.parallel.controller.dependency`:



Dependency utilities

## 8.63.2 Classes

### Dependency

```
class IPython.parallel.controller.dependency.Dependency (dependencies=[
], all=True,
success=True,
failure=False)
```

Bases: set

An object for representing a set of msg\_id dependencies.

Subclassed from set().

**Parameters dependencies:** list/set of msg\_ids or AsyncResult objects or output of Dependency.as\_dict() :

The msg\_ids to depend on

**all** : bool [default True]

Whether the dependency should be considered met when *all* depending tasks have completed or only when *any* have been completed.

**success** : bool [default True]

Whether to consider successes as fulfilling dependencies.

**failure** : bool [default False]

Whether to consider failures as fulfilling dependencies.

**If ‘all=success=True’ and ‘failure=False’, then the task will fail with an ImpossibleDependency :**

as soon as the first depended-upon task fails.

**\_\_init\_\_** (dependencies=[ ], all=True, success=True, failure=False)

**add**

Add an element to a set.

This has no effect if the element is already present.

**as\_dict** ()

Represent this dependency as a dict. For json compatibility.

**check** (completed, failed=None)

check whether our dependencies have been met.

**clear**

Remove all elements from this set.

**copy**

Return a shallow copy of a set.

**difference**

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

**difference\_update**

Remove all elements of another set from this set.

**discard**

Remove an element from a set if it is a member.

If the element is not a member, do nothing.

**intersection**

Return the intersection of two sets as a new set.

(i.e. all elements that are in both sets.)

**intersection\_update**

Update a set with the intersection of itself and another.

**isdisjoint**

Return True if two sets have a null intersection.

**issubset**

Report whether another set contains this set.

**issuperset**

Report whether this set contains another set.

**pop**

Remove and return an arbitrary set element. Raises `KeyError` if the set is empty.

**remove**

Remove an element from a set; it must be a member.

If the element is not a member, raise a `KeyError`.

**symmetric\_difference**

Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

**symmetric\_difference\_update**

Update a set with the symmetric difference of itself and another.

**union**

Return the union of sets as a new set.

(i.e. all elements that are in either set.)

**unreachable** (*completed, failed=None*)

return whether this dependency has become impossible.

**update**

Update a set with the union of itself and others.

## depend

**class** IPython.parallel.controller.dependency.**depend** (*f*, \**args*, \*\**kwargs*)

Bases: object

Dependency decorator, for use with tasks.

@*depend* lets you define a function for engine dependencies just like you use *apply* for tasks.

### Examples

```
@depend(df, a,b, c=5)
def f(m,n,p)

view.apply(f, 1,2,3)
```

will call df(a,b,c=5) on the engine, and if it returns False or raises an UnmetDependency error, then the task will not be run and another engine will be tried.

```
__init__ (f, *args, **kwargs)
```

## dependent

**class** IPython.parallel.controller.dependency.**dependent** (*f*, *df*, \**dargs*, \*\**dkwargs*)

Bases: object

A function that depends on another function. This is an object to prevent the closure used in traditional decorators, which are not picklable.

```
__init__ (f, df, *dargs, **dkwargs)
```

### 8.63.3 Function

IPython.parallel.controller.dependency.**require** (\**mods*)

Simple decorator for requiring names to be importable.

### Examples

```
In [1]: @require('numpy') ...: def norm(a): ...: import numpy ...: return numpy.linalg.norm(a,2)
```

## 8.64 parallel.controller.dictdb

### 8.64.1 Module: parallel.controller.dictdb

Inheritance diagram for IPython.parallel.controller.dictdb:





A Task logger that presents our DB interface, but exists entirely in memory and implemented with dicts.

TaskRecords are dicts of the form: {

```

    'msg_id' : str(uuid), 'client_uuid' : str(uuid), 'engine_uuid' : str(uuid) or None, 'header' :
    dict(header), 'content': dict(content), 'buffers': list(buffers), 'submitted': datetime, 'started':
    datetime or None, 'completed': datetime or None, 'resubmitted': datetime or None, 're-
    sult_header' : dict(header) or None, 'result_content' : dict(content) or None, 'result_buffers' :
    list(buffers) or None,
  
```

} With this info, many of the special categories of tasks can be defined by query:

pending: completed is None client's outstanding: client\_uuid = uuid && completed is None MIA: arrived is None (and completed is None) etc.

EngineRecords are dicts of the form: {

```

    'eid' : int(id), 'uuid': str(uuid)
  
```

} This may be extended, but is currently.

**We support a subset of mongodb operators:** \$lt,\$gt,\$lte,\$gte,\$ne,\$in,\$nin,\$all,\$mod,\$exists

## 8.64.2 Classes

### BaseDB

```
class IPython.parallel.controller.dictdb.BaseDB(**kwargs)
```

Bases: IPython.config.configurable.Configurable

Empty Parent class so traitlets work on DB.

```
__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.

**session**

A casting version of the unicode trait.

**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.

## CompositeFilter

**class** IPython.parallel.controller.dictdb.**CompositeFilter** (*dikt*)

Bases: object

Composite filter for matching multiple properties.

**\_\_init\_\_** (*dikt*)

## DictDB

**class** IPython.parallel.controller.dictdb.**DictDB** (*\*\*kwargs*)

Bases: IPython.parallel.controller.dictdb.BaseDB

Basic in-memory dict-based object for saving Task Records.

This is the first object to present the DB interface for logging tasks out of memory.

The interface is based on MongoDB, so adding a MongoDB backend should be straightforward.

**\_\_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.

**add\_record** (*msg\_id, rec*)

Add a new Task Record, by msg\_id.

**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.

**drop\_matching\_records** (*check*)

Remove a record from the DB.

**drop\_record** (*msg\_id*)

Remove a record from the DB.

**find\_records** (*check*, *id\_only=False*)

Find records matching a query dict.

**get\_record** (*msg\_id*)

Get a specific Task Record, by msg\_id.

**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.

**session**

A casting version of the unicode trait.

**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.

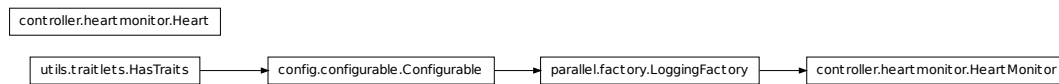
**update\_record** (*msg\_id*, *rec*)

Update the data in an existing record.

## 8.65 parallel.controller.heartmonitor

### 8.65.1 Module: parallel.controller.heartmonitor

Inheritance diagram for IPython.parallel.controller.heartmonitor:



A multi-heart Heartbeat system using PUB and XREP sockets. pings are sent out on the PUB, and hearts are tracked based on their XREQ identities.

### 8.65.2 Classes

#### Heart

```
class IPython.parallel.controller.heartmonitor.Heart (in_addr,      out_addr,
                                                    in_type=2, out_type=5,
                                                    heart_id=None)
```

Bases: object

A basic heart object for responding to a HeartMonitor. This is a simple wrapper with defaults for the most common Device model for responding to heartbeats.

It simply builds a threadsafe zmq.FORWARDER Device, defaulting to using SUB/XREQ for in/out.

You can specify the XREQ's IDENTITY via the optional heart\_id argument.

```
__init__ (in_addr, out_addr, in_type=2, out_type=5, heart_id=None)
```

```
start ()
```

#### HeartMonitor

```
class IPython.parallel.controller.heartmonitor.HeartMonitor (**kwargs)
    Bases: IPython.parallel.factory.LoggingFactory
```

A basic HeartMonitor class pingstream: a PUB stream pongstream: an XREP stream period: the period of the heartbeat in milliseconds

```
__init__ (**kwargs)
```

```
add_heart_failure_handler (handler)
    add a new handler for heart failure
```

**add\_new\_heart\_handler** (*handler*)  
add a new handler for new hearts

**beat** ()

**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.

**debug**  
A boolean (True, False) trait.

**handle\_heart\_failure** (*heart*)

**handle\_new\_heart** (*heart*)

**handle\_pong** (*msg*)  
a heart just beat

**hearts**  
An instance of a Python set.

**last\_ping**  
A casting version of the float trait.

**lifetime**  
A casting version of the float trait.

**log**  
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.

**logname**  
A casting version of the unicode trait.

**loop**  
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\_probation**  
An instance of a Python set.

**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.

**period**

A casting version of the float trait.

**pingstream**

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.

**pongstream**

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.

**responses**

An instance of a Python set.

**start** ()

**tic**

A casting version of the float trait.

**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.

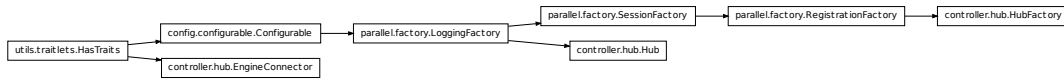
The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

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.

## 8.66 parallel.controller.hub

### 8.66.1 Module: `parallel.controller.hub`

Inheritance diagram for `IPython.parallel.controller.hub`:



The IPython Controller Hub with 0MQ This is the master object that handles connections from engines and clients, and monitors traffic through the various queues.

## 8.66.2 Classes

### EngineConnector

```
class IPython.parallel.controller.hub.EngineConnector (**kw)
```

Bases: IPython.utils.traitlets.HasTraits

A simple object for accessing the various zmq connections of an object. Attributes are: id (int): engine ID uuid (str): uuid (unused?) queue (str): identity of queue's XREQ socket registration (str): identity of registration XREQ socket heartbeat (str): identity of heartbeat XREQ socket

```
__init__ (**kw)
```

**control**

A trait for strings.

**heartbeat**

A trait for strings.

**id**

A integer 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.

#### **pending**

An instance of a Python set.

#### **queue**

A trait for strings.

#### **registration**

A trait for strings.

#### **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.

## **Hub**

**class** IPython.parallel.controller.hub.**Hub** (*\*\*kwargs*)  
 Bases: IPython.parallel.factory.LoggingFactory

The IPython Controller Hub with 0MQ connections

**Parameters** **loop**: zmq IOLoop instance :

**session**: StreamSession object :

**<removed> context**: zmq context for creating new connections (?) :

**queue**: ZMQStream for monitoring the command queue (SUB) :

**query**: ZMQStream for engine registration and client queries requests (XREP) :

**heartbeat**: HeartMonitor object checking the pulse of the engines :

**notifier**: ZMQStream for broadcasting engine registration changes (PUB) :

**db**: connection to db for out of memory logging of commands :

NotImplemented

**engine\_info**: dict of zmq connection information for engines to connect :

to the queues.

**client\_info**: dict of zmq connection information for engines to connect :

to the queues.

**\_\_init\_\_** (*\*\*kwargs*)

# universal: loop: IOLoop for creating future connections session: streamsession for sending serialized data # engine: queue: ZMQStream for monitoring queue messages query: ZMQStream for engine+client registration and client requests heartbeat: HeartMonitor object for tracking engines # extra: db: ZMQStream for db connection (NotImplemented) engine\_info: zmq address/protocol dict for engine connections client\_info: zmq address/protocol dict for client connections

**all\_completed**

An instance of a Python set.

**by\_ident**

An instance of a Python dict.

**check\_load** (*client\_id, msg*)

**client\_info**

An instance of a Python dict.

**clients**

An instance of a Python dict.

**completed**

An instance of a Python dict.

**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.

**connection\_request** (*client\_id, msg*)

Reply with connection addresses for clients.

**db**

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.

**dead\_engines**

An instance of a Python set.

**dispatch\_db** (*msg*)

**dispatch\_monitor\_traffic** (*msg*)

all ME and Task queue messages come through here, as well as IOPub traffic.

**dispatch\_query** (*msg*)

Route registration requests and queries from clients.

**engine\_info**

An instance of a Python dict.

**engines**

An instance of a Python dict.

**finish\_registration** (*heart*)

Second half of engine registration, called after our HeartMonitor has received a beat from the Engine's Heart.

**get\_results** (*client\_id, msg*)

Get the result of 1 or more messages.

**handle\_heart\_failure** (*heart*)

handler to attach to heartbeater. called when a previously registered heart fails to respond to beat request. triggers unregistration

**handle\_new\_heart** (*heart*)

handler to attach to heartbeater. Called when a new heart starts to beat. Triggers completion of registration.

**heartmonitor**

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.

**hearts**

An instance of a Python dict.

**ids**

An instance of a Python set.

**incoming\_registrations**

An instance of a Python dict.

**keytable**

An instance of a Python dict.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**mia\_task\_request** (*idents, msg*)**monitor**

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.

**notifier**

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.

**pending**

An instance of a Python set.

**purge\_results** (*client\_id, msg*)

Purge results from memory. This method is more valuable before we move to a DB based message storage mechanism.

**query**

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.

**queue\_status** (*client\_id, msg*)

Return the Queue status of one or more targets. if verbose: return the msg\_ids else: return len of each type. keys: queue (pending MUX jobs)

tasks (pending Task jobs) completed (finished jobs from both queues)

**queues**

An instance of a Python dict.

**register\_engine** (*reg, msg*)

Register a new engine.

**registration\_timeout**

A integer trait.

**resubmit\_task** (*client\_id, msg, buffers*)

Resubmit a task.

**save\_iopub\_message** (*topics, msg*)

save an iopub message into the db

**save\_queue\_request** (*idents, msg*)

**save\_queue\_result** (*idents, msg*)

**save\_task\_destination** (*idents, msg*)

**save\_task\_request** (*idents, msg*)

Save the submission of a task.

**save\_task\_result** (*idents, msg*)

save the result of a completed task.

**shutdown\_request** (*client\_id, msg*)

handle shutdown request.

**tasks**

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.

**unregister\_engine** (*ident, msg*)

Unregister an engine that explicitly requested to leave.

## HubFactory

**class** IPython.parallel.controller.hub.**HubFactory** (*\*\*kwargs*)

Bases: IPython.parallel.factory.RegistrationFactory

The Configurable for setting up a Hub.

**\_\_init\_\_** (*\*\*kwargs*)

**client\_ip**

A casting version of the string trait.

**client\_transport**

A casting version of the string trait.

**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.

**construct** ()

**construct\_hub()**  
construct

**context**  
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.

**control**  
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**  
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\_class**  
A casting version of the string trait.

**engine\_ip**  
A casting version of the string trait.

**engine\_transport**  
A casting version of the string trait.

**exec\_key**  
A casting version of the unicode trait.

**hb**  
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.

**heartmonitor**  
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.

**ident**  
A casting version of the string trait.

**iopub**  
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.

**ip**  
A trait for strings.

**log**  
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.

**logname**  
A casting version of the unicode trait.

**loop**

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.

**mon\_port**

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.

**monitor\_ip**

A casting version of the string trait.

**monitor\_transport**

A casting version of the string trait.

**monitor\_url**

A casting version of the string trait.

**mux**

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.

**notifier\_port**

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.

**packer**

A trait for strings.

**ping**

A integer trait.

**regport**

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.

**scheme**

A trait for strings.

**session**

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()****subconstructors**

An instance of a Python list.

**task**

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.

**transport**

A trait for strings.

**unpacker**

A trait for strings.

**url**

A trait for strings.

**username**

A casting version of the unicode trait.

### 8.66.3 Functions

`IPython.parallel.controller.hub.empty_record()`

Return an empty dict with all record keys.



`IPython.parallel.controller.hub.init_record(msg)`  
 Initialize a TaskRecord based on a request.

## 8.67 parallel.controller.mongodb

### 8.67.1 Module: parallel.controller.mongodb

Inheritance diagram for `IPython.parallel.controller.mongodb`:



A TaskRecord backend using mongodb

### 8.67.2 MongoDB

**class** `IPython.parallel.controller.mongodb.MongoDB(**kwargs)`

Bases: `IPython.parallel.controller.dictdb.BaseDB`

MongoDB TaskRecord backend.

**\_\_init\_\_** (*\*\*kwargs*)

**add\_record** (*msg\_id, rec*)

Add a new Task Record, by msg\_id.

**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.

**connection\_args**

An instance of a Python list.

**connection\_kwargs**

An instance of a Python dict.

**database**

A casting version of the unicode trait.

**drop\_matching\_records** (*check*)

Remove a record from the DB.

**drop\_record** (*msg\_id*)

Remove a record from the DB.

**find\_records** (*check, id\_only=False*)

Find records matching a query dict.

**get\_record**(*msg\_id*)

Get a specific Task Record, by *msg\_id*.

**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.

**session**

A casting version of the unicode trait.

**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.

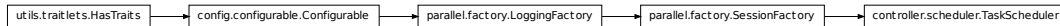
**update\_record**(*msg\_id, rec*)

Update the data in an existing record.

## 8.68 parallel.controller.scheduler

### 8.68.1 Module: parallel.controller.scheduler

Inheritance diagram for `IPython.parallel.controller.scheduler`:



The Python scheduler for rich scheduling.

The Pure ZMQ scheduler does not allow routing schemes other than LRU, nor does it check msg\_id DAG dependencies. For those, a slightly slower Python Scheduler exists.

## 8.68.2 Class

### 8.68.3 TaskScheduler

**class** IPython.parallel.controller.scheduler.**TaskScheduler** (\*\*kwargs)

Bases: IPython.parallel.factory.SessionFactory

Python TaskScheduler object.

This is the simplest object that supports msg\_id based DAG dependencies. *Only* task msg\_ids are checked, not msg\_ids of jobs submitted via the MUX queue.

**\_\_init\_\_** (\*\*kwargs)

**add\_job** (idx)

Called after self.targets[idx] just got the job with header. Override with subclasses. The default ordering is simple LRU. The default loads are the number of outstanding jobs.

**all\_completed**

An instance of a Python set.

**all\_done**

An instance of a Python set.

**all\_failed**

An instance of a Python set.

**all\_ids**

An instance of a Python set.

**audit\_timeouts** ()

Audit all waiting tasks for expired timeouts.

**auditor**

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.

**blacklist**

An instance of a Python dict.

**client\_stream**

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.

**clients**

An instance of a Python dict.

**completed**

An instance of a Python dict.

**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.

**context**

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.

**depending**

An instance of a Python dict.

**destinations**

An instance of a Python dict.

**dispatch\_notification** (*msg*)

dispatch register/unregister events.

**dispatch\_result** (*raw\_msg*)

dispatch method for result replies

**dispatch\_submission** (*raw\_msg*)

Dispatch job submission to appropriate handlers.

**engine\_stream**

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.

**exec\_key**

A casting version of the unicode trait.

**fail\_unreachable** (*msg\_id*, *why*=<class 'IPython.parallel.error.ImpossibleDependency'>)

a task has become unreachable, send a reply with an ImpossibleDependency error.

**failed**

An instance of a Python dict.

**finish\_job** (*idx*)

Called after self.targets[idx] just finished a job. Override with subclasses.

**graph**

An instance of a Python dict.

**handle\_result** (*idents*, *parent*, *raw\_msg*, *success=True*)

handle a real task result, either success or failure

**handle\_stranded\_tasks** (*engine*)

Deal with jobs resident in an engine that died.

**handle\_unmet\_dependency** (*idents, parent*)

handle an unmet dependency

**ident**

A casting version of the string trait.

**loads**

An instance of a Python list.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**maybe\_run** (*msg\_id, raw\_msg, targets, after, follow, timeout*)

check location dependencies, and run if they are met.

**mon\_stream**

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.

**notifier\_stream**

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.

**packer**

A trait for strings.

**pending**

An instance of a Python dict.

**resume\_receiving()**

Resume accepting jobs.

**save\_unmet** (*msg\_id, raw\_msg, targets, after, follow, timeout*)

Save a message for later submission when its dependencies are met.

**scheme**

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.

**session**

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()****stop\_receiving()**

Stop accepting jobs while there are no engines. Leave them in the ZMQ queue.

**submit\_task** (*msg\_id, raw\_msg, targets, follow, timeout, indices=None*)

Submit a task to any of a subset of our targets.

**targets**

An instance of a Python list.

**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.

**unpacker**

A trait for strings.

**update\_graph** (*dep\_id, success=True*)

dep\_id just finished. Update our dependency graph and submit any jobs that just became runnable.

**username**

A casting version of the unicode trait.

## 8.68.4 Functions

```
IPython.parallel.controller.scheduler.launch_scheduler(in_addr,
                                                    out_addr,
                                                    mon_addr,
                                                    not_addr,    con-
fig=None,    log-
name='ZMQ',
log_addr=None,
loglevel=10,
scheme='lru',
identity='task')
```

```
IPython.parallel.controller.scheduler.leastload(loads)
```

Always choose the lowest load.

If the lowest load occurs more than once, the first occurrence will be used. If loads has LRU ordering, this means the LRU of those with the lowest load is chosen.

```
IPython.parallel.controller.scheduler.logged(f)
```

```
IPython.parallel.controller.scheduler.lru(loads)
```

Always pick the front of the line.

The content of *loads* is ignored.

Assumes LRU ordering of loads, with oldest first.

```
IPython.parallel.controller.scheduler.plainrandom(loads)
```

Plain random pick.

```
IPython.parallel.controller.scheduler.twobin(loads)
```

Pick two at random, use the LRU of the two.

The content of loads is ignored.

Assumes LRU ordering of loads, with oldest first.

```
IPython.parallel.controller.scheduler.weighted(loads)
```

Pick two at random using inverse load as weight.

Return the less loaded of the two.

## 8.69 parallel.controller.sqlitedb

### 8.69.1 Module: parallel.controller.sqlitedb

Inheritance diagram for IPython.parallel.controller.sqlitedb:



A TaskRecord backend using sqlite3

## 8.69.2 SQLiteDB

```
class IPython.parallel.controller.sqlitedb.SQLiteDB (**kwargs)
```

Bases: `IPython.parallel.controller.dictdb.BaseDB`

SQLite3 TaskRecord backend.

```
__init__ (**kwargs)
```

```
add_record (msg_id, rec)
```

Add a new Task Record, by msg\_id.

```
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.

```
drop_matching_records (check)
```

Remove a record from the DB.

```
drop_record (msg_id)
```

Remove a record from the DB.

```
filename
```

A casting version of the unicode trait.

```
find_records (check, id_only=False)
```

Find records matching a query dict.

```
get_record (msg_id)
```

Get a specific Task Record, by msg\_id.

```
location
```

A casting version of the unicode 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.

#### **session**

A casting version of the unicode trait.

#### **table**

A casting version of the unicode trait.

#### **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.

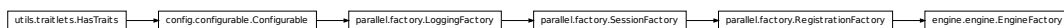
#### **update\_record** (*msg\_id, rec*)

Update the data in an existing record.

## 8.70 parallel.engine.engine

### 8.70.1 Module: `parallel.engine.engine`

Inheritance diagram for `IPython.parallel.engine.engine`:



A simple engine that talks to a controller over OMQ. it handles registration, etc. and launches a kernel connected to the Controller's Schedulers.

### 8.70.2 EngineFactory

**class** IPython.parallel.engine.engine.**EngineFactory** (\*\*kwargs)

Bases: IPython.parallel.factory.RegistrationFactory

IPython engine

**\_\_init\_\_** (\*\*kwargs)

**abort** ()

**complete\_registration** (msg)

**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.

**context**

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\_hook\_factory**

A trait whose value must be a subclass of a specified class.

**exec\_key**

A casting version of the unicode trait.

**id**

A integer trait.

**ident**

A casting version of the string trait.

**ip**

A trait for strings.

**kernel**

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.

**location**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**out\_stream\_factory**

A trait whose value must be a subclass of a specified class.

**packer**

A trait for strings.

**register** ()

send the registration\_request

**registrar**

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.

**regport**

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.

**session**

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** ()

**timeout**

A casting version of the float trait.

**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.

**transport**

A trait for strings.

**unpacker**

A trait for strings.

**url**

A trait for strings.

**user\_ns**

An instance of a Python dict.

**username**

A casting version of the unicode trait.

## 8.71 parallel.engine.kernelstarter

### 8.71.1 Module: parallel.engine.kernelstarter

Inheritance diagram for IPython.parallel.engine.kernelstarter:

engine.kernelstarter.KernelStarter

KernelStarter class that intercepts Control Queue messages, and handles process management.

### 8.71.2 KernelStarter

```
class IPython.parallel.engine.kernelstarter.KernelStarter(session,      up-
                                                             stream,  down-
                                                             stream,  *ker-
                                                             nel_args, **ker-
                                                             nel_kwargs)
```

Bases: object

Object for resetting/killing the Kernel.

**\_\_init\_\_** (*session, upstream, downstream, \*kernel\_args, \*\*kernel\_kwargs*)

**dispatch\_reply** (*raw\_msg*)

**dispatch\_request** (*raw\_msg*)

**has\_kernel**

Returns whether a kernel process has been specified for the kernel manager.

**interrupt\_kernel** ()

Interrupts the kernel. Unlike `signal_kernel`, this operation is well supported on all platforms.

**is\_alive**

Is the kernel process still running?

**kill\_kernel** ()

Kill the running kernel.

**restart\_kernel** (*now=False*)

Restarts a kernel with the same arguments that were used to launch it. If the old kernel was launched with random ports, the same ports will be used for the new kernel.

**Parameters** **now** : bool, optional

If True, the kernel is forcefully restarted *immediately*, without having a chance to do any cleanup action. Otherwise the kernel is given 1s to clean up before a forceful restart is issued.

In all cases the kernel is restarted, the only difference is whether it is given a chance to perform a clean shutdown or not.

**shutdown\_kernel** (*restart=False*)

Attempts to stop the kernel process cleanly. If the kernel cannot be stopped, it is killed, if possible.

**shutdown\_request** (*msg*)

**signal\_kernel** (*signum*)

Sends a signal to the kernel. Note that since only SIGTERM is supported on Windows, this function is only useful on Unix systems.

**start** ()

**start\_kernel** (*\*\*kw*)

Starts a kernel process and configures the manager to use it.

If random ports (`port=0`) are being used, this method must be called before the channels are created.

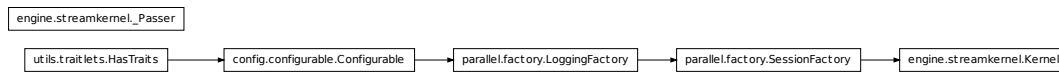
IPython.parallel.engine.kernelstarter.**make\_starter** (*up\_addr, down\_addr, \*args, \*\*kwargs*)

entry point function for launching a kernelstarter in a subprocess

## 8.72 parallel.engine.streamkernel

### 8.72.1 Module: `parallel.engine.streamkernel`

Inheritance diagram for `IPython.parallel.engine.streamkernel`:



Kernel adapted from `kernel.py` to use ZMQ Streams

### 8.72.2 Kernel

```
class IPython.parallel.engine.streamkernel.Kernel (**kwargs)
```

Bases: `IPython.parallel.factory.SessionFactory`

**\_\_init\_\_** (\*\*kwargs)

**abort\_queue** (stream)

**abort\_queues** ()

**abort\_request** (stream, ident, parent)  
abort a specifig msg by id

**aborted**  
An instance of a Python set.

**apply\_request** (stream, ident, parent)

**check\_aborted** (msg\_id)

**check\_dependencies** (dependencies)

**clear\_request** (stream, idents, parent)  
Clear our namespace.

**client**  
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.

**compiler**  
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.

**complete** (msg)

**complete\_request** (stream, ident, parent)

**completer**

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.

**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.

**context**

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.

**control\_handlers**

An instance of a Python dict.

**control\_stream**

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.

**dispatch\_control** (*msg*)**dispatch\_queue** (*stream, msg*)**exec\_key**

A casting version of the unicode trait.

**exec\_lines**

An instance of a Python list.

**execute\_request** (*stream, ident, parent*)**ident**

A casting version of the string trait.

**int\_id**

A integer trait.

**iopub\_stream**

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.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**packer**

A trait for strings.

**session**

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\_handlers**

An instance of a Python dict.

**shell\_streams**

An instance of a Python list.

**shutdown\_request** (*stream, ident, parent*)

kill myself. This should really be handled in an external process

**start** ()

**task\_stream**

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.

**unpacker**

A trait for strings.

**user\_ns**

An instance of a Python dict.

**username**

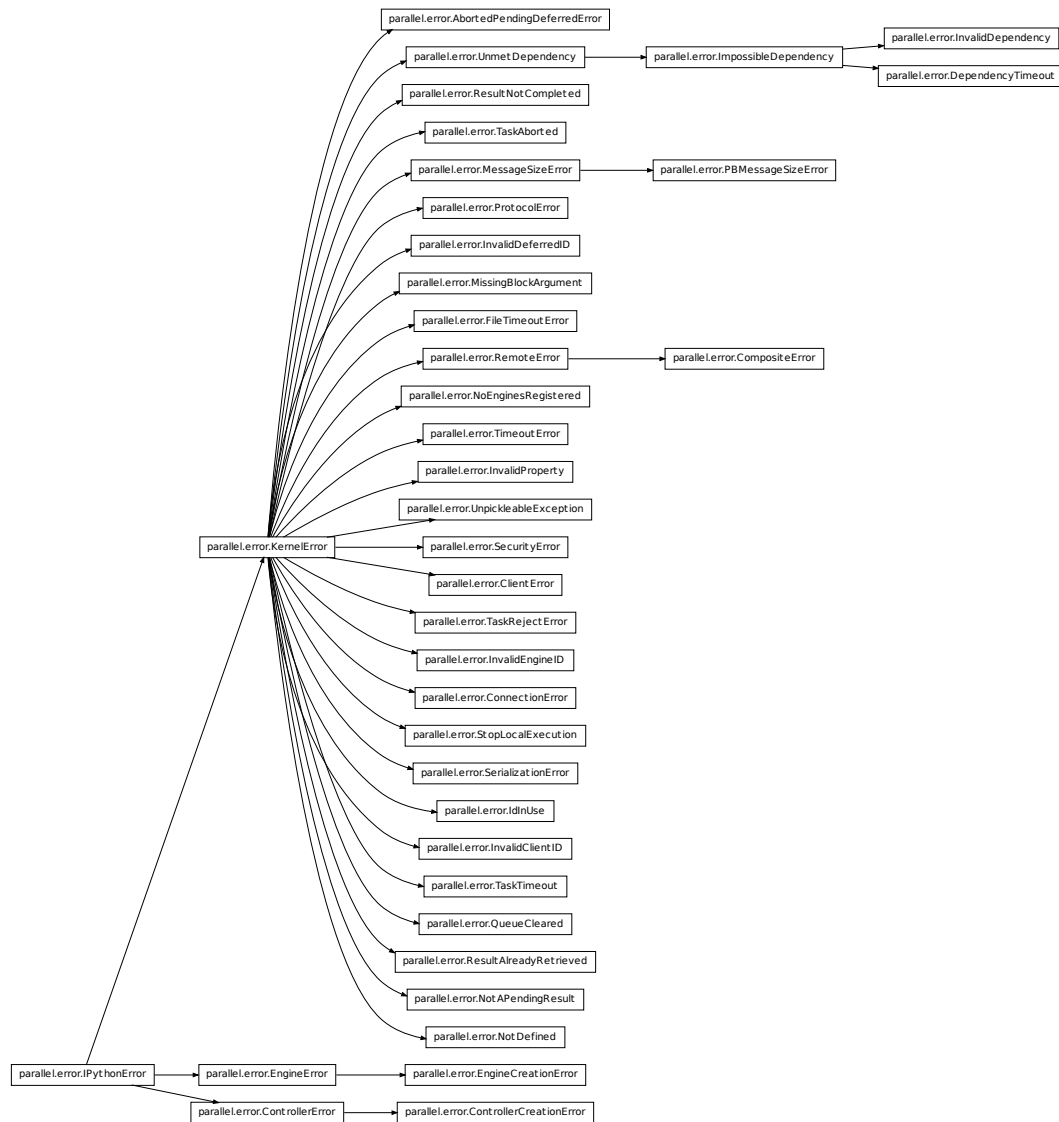
A casting version of the unicode trait.

`IPython.parallel.engine.streamkernel.printer(*args)`

## 8.73 parallel.error

### 8.73.1 Module: `parallel.error`

Inheritance diagram for `IPython.parallel.error`:



Classes and functions for kernel related errors and exceptions.

## 8.73.2 Classes

### AbortedPendingDeferredError

```
class IPython.parallel.error.AbortedPendingDeferredError
```

```
    Bases: IPython.parallel.error.KernelError
```

```
    __init__()
```

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

**args**

**message**

### ClientError

**class** `IPython.parallel.error.ClientError`

Bases: `IPython.parallel.error.KernelError`

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

**args**

**message**

### CompositeError

**class** `IPython.parallel.error.CompositeError` (*message, elist*)

Bases: `IPython.parallel.error.RemoteError`

Error for representing possibly multiple errors on engines

`__init__` (*message, elist*)

**args**

**message**

**print\_tracebacks** (*excid=None*)

**raise\_exception** (*excid=0*)

### ConnectionError

**class** `IPython.parallel.error.ConnectionError`

Bases: `IPython.parallel.error.KernelError`

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

**args**

**message**

### ControllerCreationError

**class** `IPython.parallel.error.ControllerCreationError`

Bases: `IPython.parallel.error.ControllerError`

```
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message
```

### ControllerError

```
class IPython.parallel.error.ControllerError
    Bases: IPython.parallel.error.IPythonError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

### DependencyTimeout

```
class IPython.parallel.error.DependencyTimeout
    Bases: IPython.parallel.error.ImpossibleDependency

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

### EngineCreationError

```
class IPython.parallel.error.EngineCreationError
    Bases: IPython.parallel.error.EngineError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args

    message
```

### EngineError

```
class IPython.parallel.error.EngineError
    Bases: IPython.parallel.error.IPythonError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
```

**message**

### **FileTimeoutError**

**class** IPython.parallel.error.**FileTimeoutError**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### **IPythonError**

**class** IPython.parallel.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**

### **IdInUse**

**class** IPython.parallel.error.**IdInUse**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### **ImpossibleDependency**

**class** IPython.parallel.error.**ImpossibleDependency**

Bases: IPython.parallel.error.UnmetDependency

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

#### **InvalidClientID**

**class** IPython.parallel.error.InvalidClientID

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

#### **InvalidDeferredID**

**class** IPython.parallel.error.InvalidDeferredID

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

#### **InvalidDependency**

**class** IPython.parallel.error.InvalidDependency

Bases: IPython.parallel.error.ImpossibleDependency

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

#### **InvalidEngineID**

**class** IPython.parallel.error.InvalidEngineID

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### InvalidProperty

```
class IPython.parallel.error.InvalidProperty
    Bases: IPython.parallel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

### KernelError

```
class IPython.parallel.error.KernelError
    Bases: IPython.parallel.error.IPythonError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

### MessageSizeError

```
class IPython.parallel.error.MessageSizeError
    Bases: IPython.parallel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

### MissingBlockArgument

```
class IPython.parallel.error.MissingBlockArgument
    Bases: IPython.parallel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

**NoEnginesRegistered**

```
class IPython.parallel.error.NoEnginesRegistered
    Bases: IPython.parallel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

**NotAPendingResult**

```
class IPython.parallel.error.NotAPendingResult
    Bases: IPython.parallel.error.KernelError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

**NotDefined**

```
class IPython.parallel.error.NotDefined(name)
    Bases: IPython.parallel.error.KernelError

    __init__(name)

    args
    message
```

**PBMessageSizeError**

```
class IPython.parallel.error.PBMessageSizeError
    Bases: IPython.parallel.error.MessageSizeError

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

**ProtocolError**

```
class IPython.parallel.error.ProtocolError
    Bases: IPython.parallel.error.KernelError
```



```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
args  
message
```

### QueueCleared

```
class IPython.parallel.error.QueueCleared  
    Bases: IPython.parallel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

### RemoteError

```
class IPython.parallel.error.RemoteError(ename,      evaluate,      traceback,      en-  
                                         gine_info=None)  
    Bases: IPython.parallel.error.KernelError  
  
    Error raised elsewhere  
  
    __init__(ename, evaluate, traceback, engine_info=None)  
  
    args  
  
    message
```

### ResultAlreadyRetrieved

```
class IPython.parallel.error.ResultAlreadyRetrieved  
    Bases: IPython.parallel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

### ResultNotCompleted

```
class IPython.parallel.error.ResultNotCompleted  
    Bases: IPython.parallel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature
```

**args**

**message**

### **SecurityError**

**class** IPython.parallel.error.**SecurityError**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### **SerializationError**

**class** IPython.parallel.error.**SerializationError**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### **StopLocalExecution**

**class** IPython.parallel.error.**StopLocalExecution**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### **TaskAborted**

**class** IPython.parallel.error.**TaskAborted**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_**()

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### TaskRejectError

**class** IPython.parallel.error.**TaskRejectError**

Bases: IPython.parallel.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.parallel.error.**TaskTimeout**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_()**

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### TimeoutError

**class** IPython.parallel.error.**TimeoutError**

Bases: IPython.parallel.error.KernelError

**\_\_init\_\_()**

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

### UnmetDependency

**class** IPython.parallel.error.**UnmetDependency**

Bases: IPython.parallel.error.KernelError

```
__init__()  
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
args  
  
message
```

### UnpickleableException

```
class IPython.parallel.error.UnpickleableException  
    Bases: IPython.parallel.error.KernelError  
  
    __init__()  
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature  
  
    args  
  
    message
```

## 8.73.3 Functions

`IPython.parallel.error.collect_exceptions` (*rdict\_or\_list*, *method='unspecified'*)  
 check a result dict for errors, and raise `CompositeError` if any exist. Passthrough otherwise.

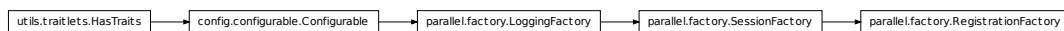
`IPython.parallel.error.unwrap_exception` (*content*)

`IPython.parallel.error.wrap_exception` (*engine\_info={}*)

## 8.74 parallel.factory

### 8.74.1 Module: parallel.factory

Inheritance diagram for `IPython.parallel.factory`:



Base config factories.

### 8.74.2 Classes

#### LoggingFactory

```
class IPython.parallel.factory.LoggingFactory (**kwargs)  
    Bases: IPython.config.configurable.Configurable
```

A most basic class, that has a *log* (type:*Logger*) attribute, set via a *logname* Trait.

**\_\_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.

## log

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.

## logname

A casting version of the unicode 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.

**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.

## RegistrationFactory

**class** IPython.parallel.factory.**RegistrationFactory** (\*\**kwargs*)

Bases: IPython.parallel.factory.SessionFactory

The Base Configurable for objects that involve registration.

**\_\_init\_\_** (\*\**kwargs*)

**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.

**context**

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.

**exec\_key**

A casting version of the unicode trait.

**ident**

A casting version of the string trait.

**ip**

A trait for strings.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**packer**

A trait for strings.

**regport**

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.

**session**

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.

**transport**

A trait for strings.

**unpacker**

A trait for strings.

**url**

A trait for strings.

**username**

A casting version of the unicode trait.

**SessionFactory**

```
class IPython.parallel.factory.SessionFactory(**kwargs)
```

```
    Bases: IPython.parallel.factory.LoggingFactory
```

The Base factory from which every factory in IPython.parallel inherits

```
    __init__(**kwargs)
```

**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.

**context**

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.

**exec\_key**

A casting version of the unicode trait.

**ident**

A casting version of the string trait.

**log**

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.

**logname**

A casting version of the unicode trait.

**loop**

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.

**packer**

A trait for strings.

**session**

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.

**unpacker**

A trait for strings.

**username**

A casting version of the unicode trait.

### 8.74.3 Functions

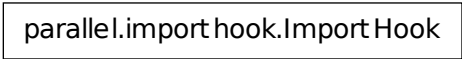
`IPython.parallel.factory.add_registration_arguments (parser)`

`IPython.parallel.factory.add_session_arguments (parser)`

## 8.75 parallel.importhook

### 8.75.1 Module: `parallel.importhook`

Inheritance diagram for `IPython.parallel.importhook`:



```
graph TD; A[parallel.importhook.ImportHook]
```

A diagram showing a single box labeled `parallel.importhook.ImportHook`.

Simple import hook for importing modules remotely via

### 8.75.2 `ImportHook`

**class** `IPython.parallel.importhook.ImportHook` (*view*)

Bases: `object`

**\_\_init\_\_** (*view*)

**find\_module** (*fullname, path=None*)

This is called by the system to find a module. `cloud.*` modules are hooked into our system

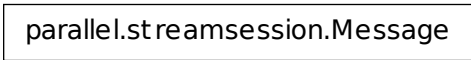
**load\_module** (*fullname*)

This is called by the system to load a module We use it to track modules after their code is loaded

## 8.76 parallel.streamsession

### 8.76.1 Module: `parallel.streamsession`

Inheritance diagram for `IPython.parallel.streamsession`:



```
graph TD; A[parallel.streamsession.Message]
```

A diagram showing a single box labeled `parallel.streamsession.Message`.



```
graph TD; A[parallel.streamsession.StreamSession]
```

A diagram showing a single box labeled `parallel.streamsession.StreamSession`.

edited session.py to work with streams, and move msg\_type to the header

## 8.76.2 Classes

### Message

**class** IPython.parallel.streamsession.**Message** (*msg\_dict*)

Bases: object

A simple message object that maps dict keys to attributes.

A Message can be created from a dict and a dict from a Message instance simply by calling dict(msg\_obj).

**\_\_init\_\_** (*msg\_dict*)

### StreamSession

**class** IPython.parallel.streamsession.**StreamSession** (*username=None, session=None, packer=None, unpacker=None, key=None, keyfile=None*)

Bases: object

tweaked version of IPython.zmq.session.Session, for development in Parallel

**\_\_init\_\_** (*username=None, session=None, packer=None, unpacker=None, key=None, keyfile=None*)

**check\_key** (*msg\_or\_header*)

Check that a message's header has the right key

**feed\_identities** (*msg, copy=True*)

feed until DELIM is reached, then return the prefix as idents and remainder as msg. This is easily broken by setting an IDENT to DELIM, but that would be silly.

**Parameters** **msg** : a list of Message or bytes objects

the message to be split

**copy** : bool

flag determining whether the arguments are bytes or Messages

**Returns** (**idents,msg**) : two lists

idents will always be a list of bytes - the identity prefix msg will be a list of bytes or Messages, unchanged from input msg should be unpackable via self.unpack\_message at this point.

**msg** (*msg\_type, content=None, parent=None, subheader=None*)

**msg\_header** (*msg\_type*)

**recv** (*socket, mode=1, content=True, copy=True*)  
receives and unpacks a message returns [idents], msg

**send** (*stream, msg\_or\_type, content=None, buffers=None, parent=None, subheader=None, ident=None, track=False*)  
Build and send a message via stream or socket.

**Parameters** **stream** : zmq.Socket or ZMQStream

the socket-like object used to send the data

**msg\_or\_type** : str or Message/dict

Normally, msg\_or\_type will be a msg\_type unless a message is being sent more than once.

**content** : dict or None

the content of the message (ignored if msg\_or\_type is a message)

**buffers** : list or None

the already-serialized buffers to be appended to the message

**parent** : Message or dict or None

the parent or parent header describing the parent of this message

**subheader** : dict or None

extra header keys for this message's header

**ident** : bytes or list of bytes

the zmq.IDENTITY routing path

**track** : bool

whether to track. Only for use with Sockets, because ZMQStream objects cannot track messages.

**Returns** **msg** : message dict

the constructed message

**(msg,tracker)** : (message dict, MessageTracker)

if track=True, then a 2-tuple will be returned, the first element being the constructed message, and the second being the MessageTracker

**send\_raw** (*stream, msg, flags=0, copy=True, ident=None*)  
Send a raw message via ident path.

**Parameters** **msg** : list of sendable buffers

**unpack\_message** (*msg, content=True, copy=True*)  
Return a message object from the format sent by self.send.

### 8.76.3 Functions

`IPython.parallel.streamsession.extract_header(msg_or_header)`

Given a message or header, return the header.

`IPython.parallel.streamsession.msg_header(msg_id, msg_type, username, session)`

`IPython.parallel.streamsession.squash_unicode(obj)`

`IPython.parallel.streamsession.test_msg2obj()`

## 8.77 parallel.util

### 8.77.1 Module: parallel.util

Inheritance diagram for `IPython.parallel.util`:

`parallel.util.ReverseDict`

`parallel.util.Namespace`

some generic utilities for dealing with classes, urls, and serialization

### 8.77.2 Classes

#### Namespace

**class** `IPython.parallel.util.Namespace`

Bases: `dict`

Subclass of `dict` for attribute access to keys.

**\_\_init\_\_()**

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

**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

**ReverseDict**

**class** IPython.parallel.util.**ReverseDict** (\*args, \*\*kwargs)

Bases: dict

simple double-keyed subset of dict methods.

**\_\_init\_\_** (\*args, \*\*kwargs)

**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 (key, default=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**

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

### 8.77.3 Functions

IPython.parallel.util.**connect\_engine\_logger** (context, iface, engine, loglevel=10)

IPython.parallel.util.**connect\_logger** (logname, context, iface, root='ip', loglevel=10)

IPython.parallel.util.**disambiguate\_ip\_address** (ip, location=None)

turn multi-ip interfaces '0.0.0.0' and '\*' into connectable ones, based on the location (default interpretation of location is localhost).

`IPython.parallel.util.disambiguate_url(url, location=None)`

turn multi-ip interfaces '0.0.0.0' and '\*' into connectable ones, based on the location (default interpretation is localhost).

This is for zeromq urls, such as tcp://\*:10101.

`IPython.parallel.util.generate_exec_key(keyfile)`

`IPython.parallel.util.integer_loglevel(loglevel)`

`IPython.parallel.util.interactive(f)`

decorator for making functions appear as interactively defined. This results in the function being linked to the user\_ns as globals() instead of the module globals().

`IPython.parallel.util.local_logger(logname, loglevel=10)`

`IPython.parallel.util.pack_apply_message(f, args, kwargs, threshold=6.399999999999997e-05)`

pack up a function, args, and kwargs to be sent over the wire as a series of buffers. Any object whose data is larger than *threshold* will not have their data copied (currently only numpy arrays support zero-copy)

`IPython.parallel.util.rekey(dikt)`

Rekey a dict that has been forced to use str keys where there should be ints by json. This belongs in the jsonutil added by fperez.

`IPython.parallel.util.select_random_ports(n)`

Selects and return *n* random ports that are available.

`IPython.parallel.util.serialize_object(obj, threshold=6.399999999999997e-05)`

Serialize an object into a list of sendable buffers.

**Parameters** *obj*: object

The object to be serialized

**threshold**: float

The threshold for not double-pickling the content.

**Returns** ('pmd', [bufs]):

where pmd is the pickled metadata wrapper, bufs is a list of data buffers

`IPython.parallel.util.signal_children(children)`

Relay interrupt/term signals to children, for more solid process cleanup.

`IPython.parallel.util.split_url(url)`

split a zmq url (tcp://ip:port) into ('tcp', 'ip', 'port').

`IPython.parallel.util.unpack_apply_message(bufs, g=None, copy=True)`

unpack f,args,kwargs from buffers packed by pack\_apply\_message() Returns: original f,args,kwargs

`IPython.parallel.util.unserialize_object(bufs)`

reconstruct an object serialized by serialize\_object from data buffers.



`IPython.parallel.util.validate_url(url)`  
validate a url for zeromq

`IPython.parallel.util.validate_url_container(container)`  
validate a potentially nested collection of urls.

## 8.78 testing

### 8.78.1 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.

## 8.79 testing.decorators

### 8.79.1 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. Using the machinery in `IPython.external.decorators`, we import either `numpy.testing.decorators` if `numpy` is available, OR use equivalent code in `IPython.external._decorators`, which we've copied verbatim from `numpy`.

## Authors

- Fernando Perez <[Fernando.Perez@berkeley.edu](mailto:Fernando.Perez@berkeley.edu)>

## 8.79.2 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.module_not_available(module)`

Can module be imported? Returns true if module does NOT import.

This is used to make a decorator to skip tests that require module 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.

## 8.80 testing.globalipapp

### 8.80.1 Module: `testing.globalipapp`

Inheritance diagram for `IPython.testing.globalipapp`:

testing.globalipapp.py\_file\_finder

testing.globalipapp.ipnsdict

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.

## 8.80.2 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

**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** (*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)
```

### 8.80.3 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.

## 8.81 testing.iptest

### 8.81.1 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) 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.

## 8.81.2 Class

### 8.81.3 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

## 8.81.4 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_ipctest()`

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_ipctestall()`

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, min_version=None)`

Test to see if mod is importable.

## 8.82 testing.ipunittest

### 8.82.1 Module: testing.ipunittest

Inheritance diagram for `IPython.testing.ipunittest`:

```
graph TD
    A[testing.ipunittest.IPython2PythonConverter]
    B[testing.ipunittest.Doc2UnitTester]
    A --> B
```

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](mailto:Fernando.Perez@berkeley.edu)>

### 8.82.2 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\_\_** ()

### 8.82.3 Functions

`IPython.testing.ipunittest.count_failures` (*runner*)

Count number of failures in a doctest runner.

Code modeled after the `summarize()` method in doctest.

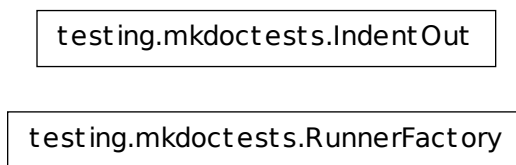


`IPython.testing.ipunittest.ipdocstring` (*func*)  
Change the function docstring via ip2py.

## 8.83 testing.mkdoctests

### 8.83.1 Module: `testing.mkdoctests`

Inheritance diagram for `IPython.testing.mkdoctests`:



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'`.

## 8.83.2 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.

## 8.83.3 Function

IPython.testing.mkdoctests.**main** ()

Run as a script.

## 8.84 testing.nosepatch

### 8.84.1 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

## 8.85 testing.plugin.dtexample

### 8.85.1 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.

### 8.85.2 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.dtexample.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.dtexample.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.dtexample.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.dtexample.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.dtxample.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’`

## 8.86 testing.plugin.show\_refs

### 8.86.1 Module: `testing.plugin.show_refs`

Inheritance diagram for `IPython.testing.plugin.show_refs`:

```
plugin.show_refs.C
```

Simple script to show reference holding behavior.

This is used by a companion test case.

### 8.86.2 C

**class** `IPython.testing.plugin.show_refs.C`

Bases: `object`

**\_\_init\_\_**()

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

## 8.87 testing.plugin.simple

### 8.87.1 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.

### 8.87.2 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
```

## 8.88 testing.plugin.test\_ipdoctest

### 8.88.1 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).

### 8.88.2 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

## 8.89 testing.plugin.test\_refs

### 8.89.1 Module: testing.plugin.test\_refs

Some simple tests for the plugin while running scripts.

### 8.89.2 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.
```

## 8.90 testing.tools

### 8.90.1 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](mailto:Fernando.Perez@berkeley.edu)>

### 8.90.2 Class

#### 8.90.3 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()
```

## 8.90.4 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. :

## 8.91 utils.PyColorize

### 8.91.1 Module: `utils.PyColorize`

Inheritance diagram for `IPython.utils.PyColorize`:

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](mailto: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).

## 8.91.2 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.

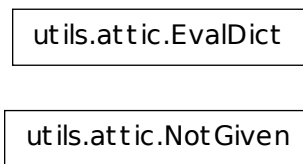
Inputs:

- argv*(None): a list of strings like `sys.argv[1:]` giving the command-line arguments. If None, use `sys.argv[1:]`.

## 8.92 `utils.attic`

### 8.92.1 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`.

### 8.92.2 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`

### 8.92.3 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 0x1082a62f0>)`

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>.

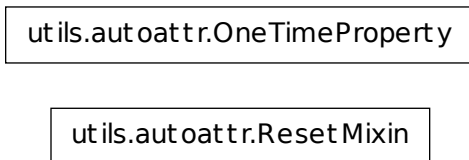
NOTE: up until IPython 0.7.2, this was called simply 'with', but 'with' has become a keyword for Python 2.5, so we had to rename it.

`IPython.utils.attic.wrap_deprecated(func, suggest='<nothing>')`

## 8.93 `utils.autoattr`

### 8.93.1 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.

## 8.93.2 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.

## 8.93.3 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
```

## 8.94 utils.codeutil

### 8.94.1 Module: `utils.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

### 8.94.2 Functions

`IPython.utils.codeutil.code_ctor(*args)`

`IPython.utils.codeutil.reduce_code(co)`

## 8.95 utils.coloransi

### 8.95.1 Module: `utils.coloransi`

Inheritance diagram for `IPython.utils.coloransi`:



utils.coloransi.ColorSchemeTable

utils.coloransi.TermColors

utils.coloransi.InputTermColors

utils.coloransi.ColorScheme

Tools for coloring text in ANSI terminals.

## 8.95.2 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 methods.

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

**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.

**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.

### 8.95.3 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.

## 8.96 utils.data

### 8.96.1 Module: utils.data

Utilities for working with data structures like lists, dicts and tuples.

### 8.96.2 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 scalars).

`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.

## 8.97 utils.decorators

### 8.97.1 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.

## 8.98 utils.dir2

### 8.98.1 Module: `utils.dir2`

A fancy version of Python's builtin `dir()` function.

### 8.98.2 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)`

## 8.99 utils.doctestreload

### 8.99.1 Module: `utils.doctestreload`

A utility for handling the reloading of doctest.

### 8.99.2 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.

## 8.100 utils.frame

### 8.100.1 Module: `utils.frame`

Utilities for working with stack frames.

### 8.100.2 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.

## 8.101 `utils.generics`

### 8.101.1 Module: `utils.generics`

Generic functions for extending IPython.

See <http://cheeseshop.python.org/pypi/simplegeneric>.

### 8.101.2 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?`

## 8.102 `utils.growl`

### 8.102.1 Module: `utils.growl`

Inheritance diagram for `IPython.utils.growl`:

utils.growl.Notifier

utils.growl.IPythonGrowlError

Utilities using Growl on OS X for notifications.

## 8.102.2 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*)

## 8.102.3 Functions

IPython.utils.growl.**notify**(*title*, *msg*)

IPython.utils.growl.**notify\_deferred**(*r*, *msg*)

IPython.utils.growl.**start**(*app\_name*)



## 8.103 `utils.importstring`

### 8.103.1 Module: `utils.importstring`

A simple utility to import something by its string name.

Authors:

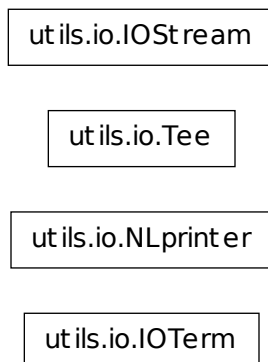
- Brian Granger

`IPython.utils.importstring.import_item(name)`  
Import and return bar given the string `foo.bar`.

## 8.104 `utils.io`

### 8.104.1 Module: `utils.io`

Inheritance diagram for `IPython.utils.io`:



IO related utilities.

### 8.104.2 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 given, 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.

### 8.104.3 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.

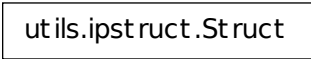
**Returns** (**filename**, **open filehandle**) :

It is the caller's responsibility to close the open file and unlink it.

## 8.105 utils.ipstruct

### 8.105.1 Module: `utils.ipstruct`

Inheritance diagram for `IPython.utils.ipstruct`:



```
graph TD; A[utils.ipstruct.Struct]
```

A dict subclass that supports attribute style access.

Authors:

- Fernando Perez (original)
- Brian Granger (refactoring to a dict subclass)

### 8.105.2 `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

### **itervalues**

D.itervalues() -> 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

## 8.106 utils.jsonutil

### 8.106.1 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

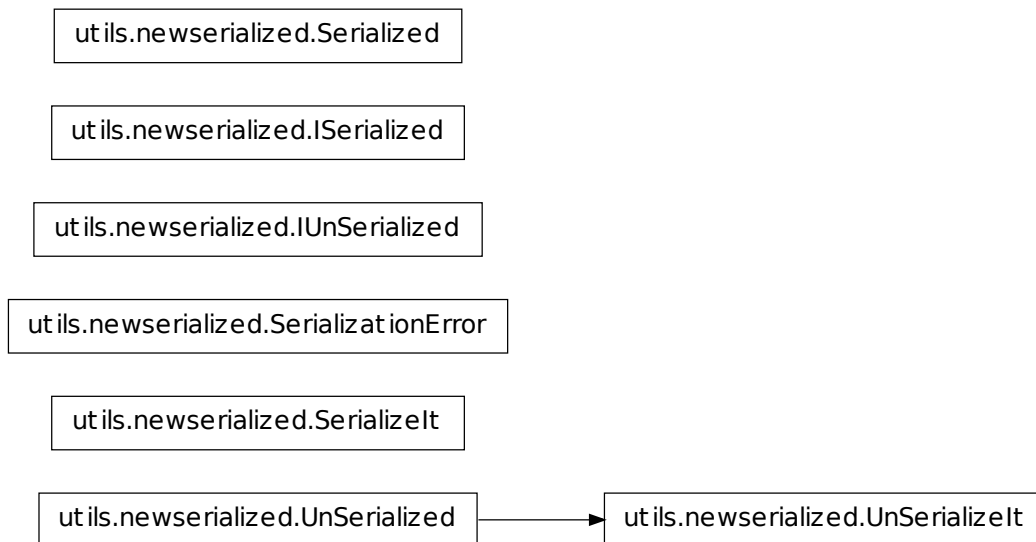
```
>>> json_clean(4)
4
>>> json_clean(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> json_clean(dict(x=1, y=2))
{'y': 2, 'x': 1}
>>> json_clean(dict(x=1, y=2, z=[1, 2, 3]))
{'y': 2, 'x': 1, 'z': [1, 2, 3]}
>>> json_clean(True)
True
```



## 8.107 utils.newserialized

### 8.107.1 Module: `utils.newserialized`

Inheritance diagram for `IPython.utils.newserialized`:



Refactored serialization classes and interfaces.

### 8.107.2 Classes

#### **ISerialized**

```
class IPython.utils.newserialized.ISerialized
```

```

    getData ()
    getDataSize (units=1000000.0)
    getMetadata ()
    getTypeDescriptor ()
  
```

#### **IUnSerialized**

```
class IPython.utils.newserialized.IUnSerialized
```

```
getObject ()
```

### SerializationError

```
class IPython.utils.newserialized.SerializationError
    Bases: exceptions.Exception

    __init__ ()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

### SerializeIt

```
class IPython.utils.newserialized.SerializeIt (unSerialized)
    Bases: object

    __init__ (unSerialized)

    getData ()

    getDataSize (units=1000000.0)

    getMetadata ()

    getTypeDescriptor ()
```

### Serialized

```
class IPython.utils.newserialized.Serialized (data, typeDescriptor, metadata={})
    Bases: object

    __init__ (data, typeDescriptor, metadata={})

    getData ()

    getDataSize (units=1000000.0)

    getMetadata ()

    getTypeDescriptor ()
```

### UnSerializeIt

```
class IPython.utils.newserialized.UnSerializeIt (serialized)
    Bases: IPython.utils.newserialized.UnSerialized

    __init__ (serialized)

    getObject ()
```

## UnSerialized

```
class IPython.utils.newserialized.UnSerialized(obj)
    Bases: object
    __init__(obj)
    getObject()
```

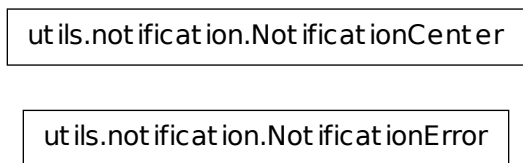
### 8.107.3 Functions

```
IPython.utils.newserialized.serialize(obj)
IPython.utils.newserialized.unserialize(serialized)
```

## 8.108 utils.notification

### 8.108.1 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

### 8.108.2 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.util.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**

## 8.109 utils.path

### 8.109.1 Module: `utils.path`

Inheritance diagram for IPython.utils.path:

utils.pat h.HomeDirError

Utilities for path handling.

### 8.109.2 Class

#### 8.109.3 HomeDirError

**class** IPython.utils.path.**HomeDirError**

Bases: exceptions.Exception

**\_\_init\_\_()**

x.\_\_init\_\_(...) initializes x; see x.\_\_class\_\_.\_\_doc\_\_ for signature

**args**

**message**

## 8.109.4 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.get_xdg_dir()`

Return the `XDG_CONFIG_HOME`, if it is defined and exists, else `None`.

This is only for posix (Linux, Unix, OS X, etc) systems.

`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.

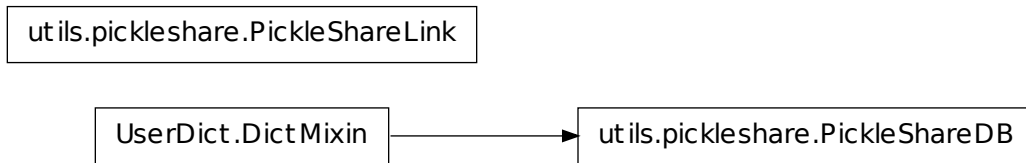
`target_update(target, deps, cmd) -> runs cmd if target is outdated.`

This is just a wrapper around `target_outdated()` which calls the given command if target is outdated.

## 8.110 utils.pickleshare

### 8.110.1 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](mailto:vivainio@gmail.com)> License: MIT open source license.

## 8.110.2 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 0x10042b850>*, *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*)

## 8.110.3 Functions

IPython.utils.pickleshare.**gethashfile** (*key*)

IPython.utils.pickleshare.**main** ()

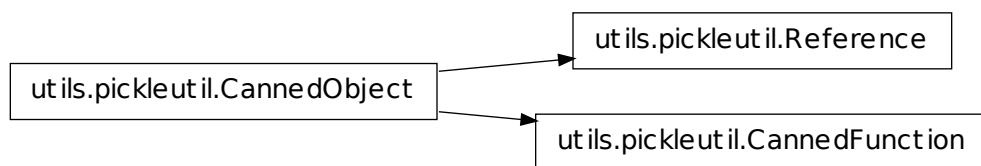
IPython.utils.pickleshare.**stress** ()

IPython.utils.pickleshare.**test** ()

## 8.111 utils.pickleutil

### 8.111.1 Module: utils.pickleutil

Inheritance diagram for IPython.utils.pickleutil:



Pickle related utilities. Perhaps this should be called ‘can’.

### 8.111.2 Classes

#### CannedFunction

**class** IPython.utils.pickleutil.**CannedFunction** (*f*)

Bases: IPython.utils.pickleutil.CannedObject

```
__init__(f)
getObject(g=None)
```

### CannedObject

```
class IPython.utils.pickleutil.CannedObject(obj, keys=[])
    Bases: object
    __init__(obj, keys=[])
    getObject(g=None)
```

### Reference

```
class IPython.utils.pickleutil.Reference(name)
    Bases: IPython.utils.pickleutil.CannedObject
    object for wrapping a remote reference by name.
    __init__(name)
    getObject(g=None)
```

## 8.111.3 Functions

```
IPython.utils.pickleutil.can(obj)
IPython.utils.pickleutil.canDict(obj)
IPython.utils.pickleutil.canSequence(obj)
IPython.utils.pickleutil.rebindFunctionGlobals(f, glbls)
IPython.utils.pickleutil.uncan(obj, g=None)
IPython.utils.pickleutil.uncanDict(obj, g=None)
IPython.utils.pickleutil.uncanSequence(obj, g=None)
```

## 8.112 utils.process

### 8.112.1 Module: utils.process

Inheritance diagram for `IPython.utils.process`:

utils.process.FindCmdError

Utilities for working with external processes.

## 8.112.2 Class

### 8.112.3 FindCmdError

```
class IPython.utils.process.FindCmdError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

## 8.112.4 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. :

## 8.113 utils.strdispatch

### 8.113.1 Module: `utils.strdispatch`

Inheritance diagram for `IPython.utils.strdispatch`:

`utils.strdispatch.StrDispatch`

String dispatch class to match regexps and dispatch commands.

### 8.113.2 `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*)

## 8.114 utils.sysinfo

### 8.114.1 Module: `utils.sysinfo`

Utilities for getting information about IPython and the system it's running in.

### 8.114.2 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.

## 8.115 `utils.syspathcontext`

### 8.115.1 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

### 8.115.2 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)
```

### `prepend_to_syspath`

```
class IPython.utils.syspathcontext.prepend_to_syspath(dir)
```

Bases: `object`

A context for prepending a directory to `sys.path` for a second.

```
__init__(dir)
```

## 8.116 utils.terminal

### 8.116.1 Module: `utils.terminal`

Utilities for working with terminals.

Authors:

- Brian E. Granger
- Fernando Perez
- Alexander Belchenko (e-mail: bialix AT ukr.net)

### 8.116.2 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`

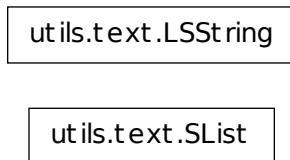
If `True`, `set_term_title()` actually writes to the terminal (using the appropriate platform-specific module). If `False`, it is a no-op.



## 8.117 utils.text

### 8.117.1 Module: `utils.text`

Inheritance diagram for `IPython.utils.text`:



Utilities for working with strings and text.

### 8.117.2 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**

### 8.117.3 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.

## 8.118 utils.timing

### 8.118.1 Module: `utils.timing`

Utilities for timing code execution.

## 8.118.2 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, out-put)`

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.

## 8.119 `utils.upgradedir`

### 8.119.1 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.

### 8.119.2 Functions

`IPython.utils.upgradedir.showdiff(old,new)`

`IPython.utils.upgradedir.upgrade_dir(srcdir, tgtdir)`

Copy over all files in `srcdir` to `tgtdir` w/ native line endings

Creates `.upgrade_report` in `tgtdir` that stores md5sums of all files to notice changed files b/w upgrades.

## 8.120 `utils.warn`

### 8.120.1 Module: `utils.warn`

Utilities for warnings. Shouldn't we just use the built in `warnings` module.

## 8.120.2 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.

## 8.121 utils.wildcard

### 8.121.1 Module: `utils.wildcard`

Support for wildcard pattern matching in object inspection.

#### Authors

- Jörgen Stenarson <[jorgen.stenarson@bostream.nu](mailto:jorgen.stenarson@bostream.nu)>
- Thomas Kluyver

### 8.121.2 Functions

`IPython.utils.wildcard.create_typedstr2type_dicts(dont_include_in_type2typestr=['lambda'])`

Return dictionaries mapping lower case typename (e.g. 'tuple') to type objects from the types package, and vice versa.

`IPython.utils.wildcard.dict_dir(obj)`

Produce a dictionary of an object's attributes. Builds on `dir2` by checking that a `getattr()` call actually succeeds.

`IPython.utils.wildcard.filter_ns(ns, name_pattern='*', type_pattern='all', ignore_case=True, show_all=True)`

Filter a namespace dictionary by name pattern and item type.

`IPython.utils.wildcard.is_type(obj, typestr_or_type)`

`is_type(obj, typestr_or_type)` verifies if `obj` is of a certain type. It can take strings or actual python types for the second argument, i.e. '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 a namespace dictionary that match `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.iciputi-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 <pythondevdang-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-hetdigitaal.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-edcn.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>
- [ZeroMQ] ZeroMQ. <http://www.zeromq.org>
- [paramiko] paramiko. <https://github.com/robey/paramiko>
- [Matplotlib] Matplotlib. <http://matplotlib.sourceforge.net>
- [PyQt] PyQt4 <http://www.riverbankcomputing.co.uk/software/pyqt/download>
- [pygments] Pygments <http://pygments.org/>
- [ZeroMQ] ZeroMQ. <http://www.zeromq.org>
- [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>>
- [OpenSSH] <<http://www.openssh.com/>>
- [Paramiko] <<http://www.lag.net/paramiko/>>
- [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](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](http://projects.scipy.org/numpy/browser/trunk/doc/EXAMPLE_DOCSTRING.rst)



# PYTHON MODULE INDEX

## i

IPython.config.loader, ??  
IPython.core.alias, ??  
IPython.core.application, ??  
IPython.core.autocall, ??  
IPython.core.builtin\_trap, ??  
IPython.core.compilerop, ??  
IPython.core.completer, ??  
IPython.core.completerlib, ??  
IPython.core.crashhandler, ??  
IPython.core.debugger, ??  
IPython.core.display, ??  
IPython.core.display\_trap, ??  
IPython.core.displayhook, ??  
IPython.core.displaypub, ??  
IPython.core.error, ??  
IPython.core.excolors, ??  
IPython.core.extensions, ??  
IPython.core.formatters, ??  
IPython.core.history, ??  
IPython.core.hooks, ??  
IPython.core.inputsplitter, ??  
IPython.core.interactiveshell, ??  
IPython.core.ipapi, ??  
IPython.core.logger, ??  
IPython.core.macro, ??  
IPython.core.magic, ??  
IPython.core.magic\_arguments, ??  
IPython.core.oinspect, ??  
IPython.core.page, ??  
IPython.core.payload, ??  
IPython.core.payloadpage, ??  
IPython.core.plugin, ??  
IPython.core.prefilter, ??  
IPython.core.prompts, ??  
IPython.core.splitinput, ??  
IPython.core.ultratb, ??  
IPython.lib.backgroundjobs, ??  
IPython.lib.clipboard, ??  
IPython.lib.deepreload, ??  
IPython.lib.demo, ??  
IPython.lib.guisupport, ??  
IPython.lib.inputhook, ??  
IPython.lib.inputhookwx, ??  
IPython.lib.irunner, ??  
IPython.lib.latextools, ??  
IPython.lib.pretty, ??  
IPython.lib.pylabtools, ??  
IPython.parallel.apps.clusterdir, ??  
IPython.parallel.apps.ipclusterapp, ??  
IPython.parallel.apps.ipcontrollerapp, ??  
IPython.parallel.apps.ipengineapp, ??  
IPython.parallel.apps.iploggerapp, ??  
IPython.parallel.apps.launcher, ??  
IPython.parallel.apps.logwatcher, ??  
IPython.parallel.apps.winhpcjob, ??  
IPython.parallel.client.asyncresult, ??  
IPython.parallel.client.client, ??  
IPython.parallel.client.map, ??  
IPython.parallel.client.remotefunction, ??  
IPython.parallel.client.view, ??  
IPython.parallel.controller.controller, ??  
IPython.parallel.controller.dependency, ??  
IPython.parallel.controller.dictdb, ??

```

IPython.parallel.controller.heartmonitor, ??
IPython.parallel.controller.hub, ??
IPython.parallel.controller.mongodb, ??
IPython.parallel.controller.scheduler, ??
IPython.parallel.controller.sqlitedb, ??
IPython.parallel.engine.engine, ??
IPython.parallel.engine.kernelstarter, ??
IPython.parallel.engine.streamkernel, ??
IPython.parallel.error, ??
IPython.parallel.factory, ??
IPython.parallel.importhook, ??
IPython.parallel.streamsession, ??
IPython.parallel.util, ??
IPython.testing, ??
IPython.testing.decorators, ??
IPython.testing.globalipapp, ??
IPython.testing.ipctest, ??
IPython.testing.ipunittest, ??
IPython.testing.mkdoctests, ??
IPython.testing.nosepatch, ??
IPython.testing.plugin.dtexample, ??
IPython.testing.plugin.show_refs, ??
IPython.testing.plugin.simple, ??
IPython.testing.plugin.test_ipdoctest, ??
IPython.testing.plugin.test_refs, ??
IPython.testing.tools, ??
IPython.utils.attic, ??
IPython.utils.autoattr, ??
IPython.utils.codeutil, ??
IPython.utils.coloransi, ??
IPython.utils.data, ??
IPython.utils.decorators, ??
IPython.utils.dir2, ??
IPython.utils.doctestreload, ??
IPython.utils.frame, ??
IPython.utils.generics, ??
IPython.utils.growl, ??
IPython.utils.importstring, ??
IPython.utils.io, ??
IPython.utils.ipstruct, ??
IPython.utils.jsonutil, ??
IPython.utils.newserialized, ??
IPython.utils.notification, ??
IPython.utils.path, ??
IPython.utils.pickleshare, ??
IPython.utils.pickleutil, ??
IPython.utils.process, ??
IPython.utils.PyColorize, ??
IPython.utils.strdispatch, ??
IPython.utils.sysinfo, ??
IPython.utils.syspathcontext, ??
IPython.utils.terminal, ??
IPython.utils.text, ??
IPython.utils.timing, ??
IPython.utils.upgradedir, ??
IPython.utils.warn, ??
IPython.utils.wildcard, ??

```