
IPython Documentation

Release 0.12.dev

The IPython Development Team

July 31, 2011

CONTENTS

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Overview | 1 |
| 1.2 | Enhanced interactive Python shell | 1 |
| 1.3 | Interactive parallel computing | 3 |
| 2 | What's new in IPython | 5 |
| 2.1 | Development version | 5 |
| 2.2 | 0.11 Series | 5 |
| 2.3 | Issues closed in the 0.11 development cycle | 18 |
| 2.4 | 0.10 series | 35 |
| 2.5 | 0.9 series | 41 |
| 2.6 | 0.8 series | 45 |
| 3 | Installation | 47 |
| 3.1 | Overview | 47 |
| 3.2 | Quickstart | 47 |
| 3.3 | Installing IPython itself | 48 |
| 3.4 | Basic optional dependencies | 49 |
| 3.5 | Dependencies for IPython.parallel (parallel computing) | 51 |
| 3.6 | Dependencies for IPython.zmq | 51 |
| 3.7 | Dependencies for ipython qtconsole (new GUI) | 51 |
| 4 | Using IPython for interactive work | 53 |
| 4.1 | Introducing IPython | 53 |
| 4.2 | IPython Tips & Tricks | 55 |
| 4.3 | IPython reference | 57 |
| 4.4 | IPython as a system shell | 81 |
| 4.5 | A Qt Console for IPython | 86 |
| 5 | Using IPython for parallel computing | 95 |
| 5.1 | Overview and getting started | 95 |
| 5.2 | Starting the IPython controller and engines | 99 |
| 5.3 | IPython's Direct interface | 110 |
| 5.4 | The IPython task interface | 123 |
| 5.5 | Using MPI with IPython | 130 |

| | | |
|----------|---|------------|
| 5.6 | IPython’s Task Database | 132 |
| 5.7 | Security details of IPython | 134 |
| 5.8 | Getting started with Windows HPC Server 2008 | 138 |
| 5.9 | Parallel examples | 147 |
| 5.10 | DAG Dependencies | 158 |
| 5.11 | Details of Parallel Computing with IPython | 163 |
| 5.12 | Transitioning from IPython.kernel to IPython.parallel | 173 |
| 6 | Configuration and customization | 179 |
| 6.1 | Overview of the IPython configuration system | 179 |
| 6.2 | IPython extensions | 186 |
| 6.3 | IPython plugins | 187 |
| 6.4 | Configuring the <code>ipython</code> command line application | 188 |
| 6.5 | Editor configuration | 190 |
| 6.6 | Outdated configuration information that might still be useful | 192 |
| 7 | IPython developer’s guide | 197 |
| 7.1 | How to contribute to IPython | 197 |
| 7.2 | Working with IPython source code | 198 |
| 7.3 | Coding guide | 210 |
| 7.4 | Documenting IPython | 213 |
| 7.5 | Testing IPython for users and developers | 215 |
| 7.6 | Releasing IPython | 222 |
| 7.7 | Development roadmap | 222 |
| 7.8 | IPython module organization | 224 |
| 7.9 | Messaging in IPython | 225 |
| 7.10 | Messaging for Parallel Computing | 242 |
| 7.11 | Connection Diagrams of The IPython ZMQ Cluster | 248 |
| 7.12 | The magic commands subsystem | 256 |
| 7.13 | Notes on code execution in <code>InteractiveShell</code> | 257 |
| 7.14 | IPython Qt interface | 258 |
| 7.15 | Porting IPython to a two process model using zeromq | 260 |
| 8 | The IPython API | 263 |
| 8.1 | <code>config.application</code> | 263 |
| 8.2 | <code>config.configurable</code> | 268 |
| 8.3 | <code>config.loader</code> | 275 |
| 8.4 | <code>core.alias</code> | 283 |
| 8.5 | <code>core.application</code> | 287 |
| 8.6 | <code>core.autocall</code> | 292 |
| 8.7 | <code>core.builtin_trap</code> | 294 |
| 8.8 | <code>core.compilerop</code> | 296 |
| 8.9 | <code>core.completer</code> | 297 |
| 8.10 | <code>core.completerlib</code> | 303 |
| 8.11 | <code>core.crashhandler</code> | 304 |
| 8.12 | <code>core.debugger</code> | 305 |
| 8.13 | <code>core.display</code> | 315 |
| 8.14 | <code>core.display_trap</code> | 316 |

| | | |
|------|-------------------------------|-----|
| 8.15 | core.displayhook | 318 |
| 8.16 | core.displaypub | 321 |
| 8.17 | core.error | 325 |
| 8.18 | core.excolors | 327 |
| 8.19 | core.extensions | 327 |
| 8.20 | core.formatters | 330 |
| 8.21 | core.history | 358 |
| 8.22 | core.hooks | 366 |
| 8.23 | core.inputsplitter | 368 |
| 8.24 | core.interactiveshell | 375 |
| 8.25 | core.ipapi | 417 |
| 8.26 | core.logger | 417 |
| 8.27 | core.macro | 419 |
| 8.28 | core.magic | 419 |
| 8.29 | core.magic_arguments | 444 |
| 8.30 | core.oinspect | 448 |
| 8.31 | core.page | 452 |
| 8.32 | core.payload | 453 |
| 8.33 | core.payloadpage | 456 |
| 8.34 | core.plugin | 456 |
| 8.35 | core.prefilter | 460 |
| 8.36 | core.profileapp | 512 |
| 8.37 | core.profiledir | 525 |
| 8.38 | core.prompts | 529 |
| 8.39 | core.shellapp | 532 |
| 8.40 | core.splitinput | 535 |
| 8.41 | core.ultratb | 536 |
| 8.42 | lib.backgroundjobs | 547 |
| 8.43 | lib.clipboard | 553 |
| 8.44 | lib.deepreload | 553 |
| 8.45 | lib.demo | 554 |
| 8.46 | lib.guisupport | 568 |
| 8.47 | lib.inpthook | 570 |
| 8.48 | lib.irunner | 573 |
| 8.49 | lib.latextools | 579 |
| 8.50 | lib.pretty | 579 |
| 8.51 | lib.pylabtools | 585 |
| 8.52 | parallel.apps.baseapp | 586 |
| 8.53 | parallel.apps.ipclusterapp | 593 |
| 8.54 | parallel.apps.ipcontrollerapp | 615 |
| 8.55 | parallel.apps.ipengineapp | 621 |
| 8.56 | parallel.apps.iploggerapp | 630 |
| 8.57 | parallel.apps.launcher | 635 |
| 8.58 | parallel.apps.logwatcher | 717 |
| 8.59 | parallel.apps.win32support | 720 |
| 8.60 | parallel.apps.winhpcjob | 721 |
| 8.61 | parallel.client.asyncresult | 739 |
| 8.62 | parallel.client.client | 743 |

| | | |
|-------|----------------------------------|-----|
| 8.63 | parallel.client.map | 752 |
| 8.64 | parallel.client.remotefunction | 753 |
| 8.65 | parallel.client.view | 756 |
| 8.66 | parallel.controller.dependency | 772 |
| 8.67 | parallel.controller.dictdb | 775 |
| 8.68 | parallel.controller.heartmonitor | 781 |
| 8.69 | parallel.controller.hub | 784 |
| 8.70 | parallel.controller.scheduler | 794 |
| 8.71 | parallel.controller.sqlitedb | 800 |
| 8.72 | parallel.engine.engine | 803 |
| 8.73 | parallel.engine.kernelstarter | 806 |
| 8.74 | parallel.engine.streamkernel | 807 |
| 8.75 | parallel.error | 811 |
| 8.76 | parallel.factory | 823 |
| 8.77 | parallel.util | 825 |
| 8.78 | testing | 829 |
| 8.79 | testing.decorators | 830 |
| 8.80 | testing.globalipapp | 832 |
| 8.81 | testing.ipptest | 835 |
| 8.82 | testing.ipunitest | 836 |
| 8.83 | testing.mkdoctests | 838 |
| 8.84 | testing.nosepatch | 840 |
| 8.85 | testing.plugin.dtexample | 840 |
| 8.86 | testing.plugin.show_refs | 842 |
| 8.87 | testing.plugin.simple | 843 |
| 8.88 | testing.plugin.test_ipdoctest | 843 |
| 8.89 | testing.plugin.test_refs | 844 |
| 8.90 | testing.skipdoctest | 845 |
| 8.91 | testing.tools | 845 |
| 8.92 | utils.PyColorize | 848 |
| 8.93 | utils.attic | 849 |
| 8.94 | utils.autoattr | 851 |
| 8.95 | utils.codeutil | 854 |
| 8.96 | utils.coloransi | 854 |
| 8.97 | utils.daemonize | 859 |
| 8.98 | utils.data | 859 |
| 8.99 | utils.decorators | 860 |
| 8.100 | utils.dir2 | 860 |
| 8.101 | utils.doctestreload | 860 |
| 8.102 | utils.frame | 861 |
| 8.103 | utils.generics | 862 |
| 8.104 | utils.growl | 863 |
| 8.105 | utils.importstring | 864 |
| 8.106 | utils.io | 864 |
| 8.107 | utils.ipstruct | 867 |
| 8.108 | utils.jsonutil | 871 |
| 8.109 | utils.newserialized | 872 |
| 8.110 | utils.notification | 875 |

| | |
|--------------------------------------|------------|
| 8.111 utils.path | 877 |
| 8.112 utils.pickleshare | 880 |
| 8.113 utils.pickleutil | 882 |
| 8.114 utils.process | 884 |
| 8.115 utils.strdispatch | 885 |
| 8.116 utils.sysinfo | 886 |
| 8.117 utils.syspathcontext | 887 |
| 8.118 utils.terminal | 888 |
| 8.119 utils.text | 889 |
| 8.120 utils.timing | 899 |
| 8.121 utils.traitslets | 900 |
| 8.122 utils.upgradedir | 935 |
| 8.123 utils.warn | 935 |
| 8.124 utils.wildcard | 936 |
| 9 About IPython | 939 |
| 9.1 Credits | 939 |
| 9.2 History | 943 |
| 9.3 License and Copyright | 944 |
| Bibliography | 947 |
| Python Module Index | 949 |
| Index | 951 |

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 can 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 Python 2.6 and 2.7. Versions 0.9 and 0.10 worked with Python 2.4 and above. IPython now also supports Python 3, although for now the code for this is separate, and kept up to date with the main IPython repository. In the future, these will converge to a single codebase which can be automatically translated using 2to3.

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

The changes listed here are a brief summary of the substantial work on IPython since the 0.11.x release series. For more details, please consult the actual source.

2.1.1 Main *ipython* branch

New features

Backwards incompatible changes

2.2 0.11 Series

2.2.1 Release 0.11

IPython 0.11 is a *major* overhaul of IPython, two years in the making. Most of the code base has been rewritten or at least reorganized, breaking backward compatibility with several APIs in previous versions. It is the first major release in two years, and probably the most significant change to IPython since its inception. We plan to have a relatively quick succession of releases, as people discover new bugs and regressions. Once we iron out any significant bugs in this process and settle down the new APIs, this series will become IPython 1.0. We encourage feedback now on the core APIs, which we hope to maintain stable during the 1.0 series.

Since the internal APIs have changed so much, projects using IPython as a library (as opposed to end-users of the application) are the most likely to encounter regressions or changes that break their existing use patterns. We will make every effort to provide updated versions of the APIs to facilitate the transition, and we encourage you to contact us on the [development mailing list](#) with questions and feedback.

Chris Fonnesbeck recently wrote an [excellent post](#) that highlights some of our major new features, with examples and screenshots. We encourage you to read it as it provides an illustrated, high-level overview complementing the detailed feature breakdown in this document.

A quick summary of the major changes (see below for details):

- **Standalone Qt console:** a new rich console has been added to IPython, started with `ipython qtconsole`. In this application we have tried to retain the feel of a terminal for fast and efficient workflows, while adding many features that a line-oriented terminal simply can not support, such as inline figures, full multiline editing with syntax highlighting, graphical tooltips for function calls and much more. This development was sponsored by [Enthought Inc.](#). See [below](#) for details.
- **High-level parallel computing with ZeroMQ.** Using the same architecture that our Qt console is based on, we have completely rewritten our high-level parallel computing machinery that in prior versions used the Twisted networking framework. While this change will require users to update their codes, the improvements in performance, memory control and internal consistency across our codebase convinced us it was a price worth paying. We have tried to explain how to best proceed with this update, and will be happy to answer questions that may arise. A full tutorial describing these features [was presented at SciPy'11](#), more details [below](#).
- **New model for GUI/plotting support in the terminal.** Now instead of the various `-Xthread` flags we had before, GUI support is provided without the use of any threads, by directly integrating GUI event loops with Python's `PyOS_InputHook` API. A new command-line flag `-gui` controls GUI support, and it can also be enabled after IPython startup via the new `%gui` magic. This requires some changes if you want to execute GUI-using scripts inside IPython, see [the GUI support section](#) for more details.
- **A two-process architecture.** The Qt console is the first use of a new model that splits IPython between a kernel process where code is executed and a client that handles user interaction. We plan on also providing terminal and web-browser based clients using this infrastructure in future releases. This model allows multiple clients to interact with an IPython process through a [well-documented messaging protocol](#) using the ZeroMQ networking library.
- **Refactoring.** the entire codebase has been refactored, in order to make it more modular and easier to contribute to. IPython has traditionally been a hard project to participate because the old codebase was very monolithic. We hope this (ongoing) restructuring will make it easier for new developers to join us.
- **Vim integration.** Vim can be configured to seamlessly control an IPython kernel, see the files in `docs/examples/vim` for the full details. This work was done by Paul Ivanov, who prepared a nice [video demonstration](#) of the features it provides.
- **Integration into Microsoft Visual Studio.** Thanks to the work of the Microsoft Python Tools for [Visual Studio](#) team, this version of IPython has been integrated into Microsoft Visual Studio's Python tools open source plug-in. [Details below](#)
- **Improved unicode support.** We closed many bugs related to unicode input.
- **Python 3.** IPython now runs on Python 3.x. See [Python 3 support](#) for details.
- **New profile model.** Profiles are now directories that contain all relevant information for that session, and thus better isolate IPython use-cases.
- **SQLite storage for history.** All history is now stored in a SQLite database, providing support for multiple simultaneous sessions that won't clobber each other as well as the ability to perform queries

on all stored data.

- **New configuration system.** All parts of IPython are now configured via a mechanism inspired by the Enthought Traits library. Any configurable element can have its attributes set either via files that now use real Python syntax or from the command-line.
- **Pasting of code with prompts.** IPython now intelligently strips out input prompts , be they plain Python ones (>>> and . . .) or IPython ones (In [N] : and “ ...:“). More details [here](#).

Authors and support

Over 60 separate authors have contributed to this release, see *below* for a full list. In particular, we want to highlight the extremely active participation of two new core team members: Evan Patterson implemented the Qt console, and Thomas Kluyver started with our Python 3 port and by now has made major contributions to just about every area of IPython.

We are also grateful for the support we have received during this development cycle from several institutions:

- Enthought Inc funded the development of our new Qt console, an effort that required developing major pieces of underlying infrastructure, which now power not only the Qt console but also our new parallel machinery. We'd like to thank Eric Jones and Travis Oliphant for their support, as well as Ilan Schnell for his tireless work integrating and testing IPython in the [Enthought Python Distribution](#).
- Nipy/NIH: funding via the [NiPy project](#) (NIH grant 5R01MH081909-02) helped us jumpstart the development of this series by restructuring the entire codebase two years ago in a way that would make modular development and testing more approachable. Without this initial groundwork, all the new features we have added would have been impossible to develop.
- Sage/NSF: funding via the grant [Sage: Unifying Mathematical Software for Scientists, Engineers, and Mathematicians](#) (NSF grant DMS-1015114) supported a meeting in spring 2011 of several of the core IPython developers where major progress was made integrating the last key pieces leading to this release.
- Microsoft's team working on [Python Tools for Visual Studio](#) developed the integraton of IPython into the Python plugin for Visual Studio 2010.
- Google Summer of Code: in 2010, we had two students developing prototypes of the new machinery that is now maturing in this release: [Omar Zapata](#) and [Gerardo Gutiérrez](#).

Development summary: moving to Git and Github

In April 2010, after [one breakage too many with bzr](#), we decided to move our entire development process to Git and Github.com. This has proven to be one of the best decisions in the project's history, as the combination of git and github have made us far, far more productive than we could be with our previous tools. We first converted our bzr repo to a git one without losing history, and a few weeks later ported all open Launchpad bugs to github issues with their comments mostly intact (modulo some formatting changes). This ensured a smooth transition where no development history or submitted bugs were lost. Feel free to use our little [Launchpad to Github issues porting script](#) if you need to make a similar transition.

These simple statistics show how much work has been done on the new release, by comparing the current code to the last point it had in common with the 0.10 series. A huge diff and ~2200 commits make up this cycle:

```
git diff $(git merge-base 0.10.2 HEAD) | wc -l  
288019
```

```
git log $(git merge-base 0.10.2 HEAD)..HEAD --oneline | wc -l  
2200
```

Since our move to github, 511 issues were closed, 226 of which were pull requests and 285 regular issues ([a full list with links](#) is available for those interested in the details). Github's pull requests are a fantastic mechanism for reviewing code and building a shared ownership of the project, and we are making enthusiastic use of it.

Note: This undercounts the number of issues closed in this development cycle, since we only moved to github for issue tracking in May 2010, but we have no way of collecting statistics on the number of issues closed in the old Launchpad bug tracker prior to that.

Qt Console

IPython now ships with a Qt application that feels very much like a terminal, but is in fact a rich GUI that runs an IPython client but supports inline figures, saving sessions to PDF and HTML, multiline editing with syntax highlighting, graphical calltips and much more:

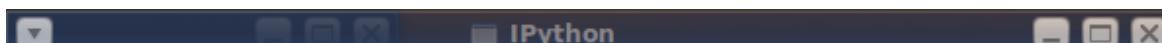
We hope that many projects will embed this widget, which we've kept deliberately very lightweight, into their own environments. In the future we may also offer a slightly more featureful application (with menus and other GUI elements), but we remain committed to always shipping this easy to embed widget.

See the [Qt console section](#) of the docs for a detailed description of the console's features and use.

High-level parallel computing with ZeroMQ

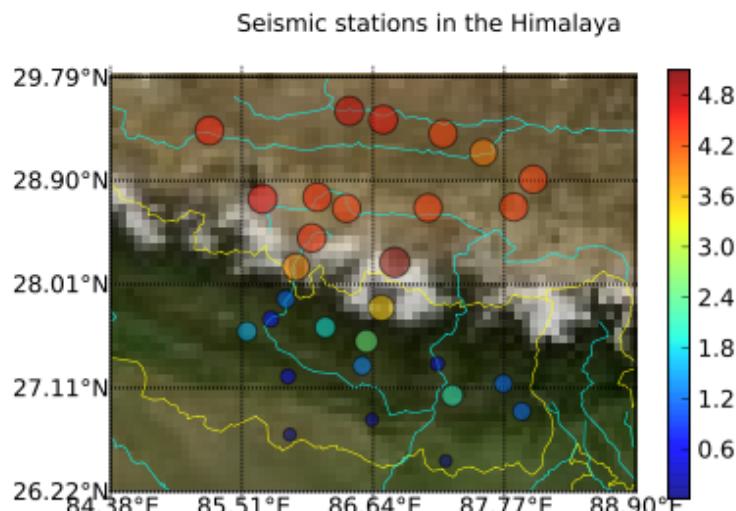
We have completely rewritten the Twisted-based code for high-level parallel computing to work atop our new ZeroMQ architecture. While we realize this will break compatibility for a number of users, we hope to make the transition as easy as possible with our docs, and we are convinced the change is worth it. ZeroMQ provides us with much tighter control over memory, higher performance, and its communications are impervious to the Python Global Interpreter Lock because they take place in a system-level C++ thread. The impact of the GIL in our previous code was something we could simply not work around, given that Twisted is itself a Python library. So while Twisted is a very capable framework, we think ZeroMQ fits our needs much better and we hope you will find the change to be a significant improvement in the long run.

Our manual contains [a full description of how to use IPython for parallel computing](#), and the [tutorial](#) presented by Min Ragan-Kelley at the SciPy 2011 conference provides a hands-on complement to the reference docs.

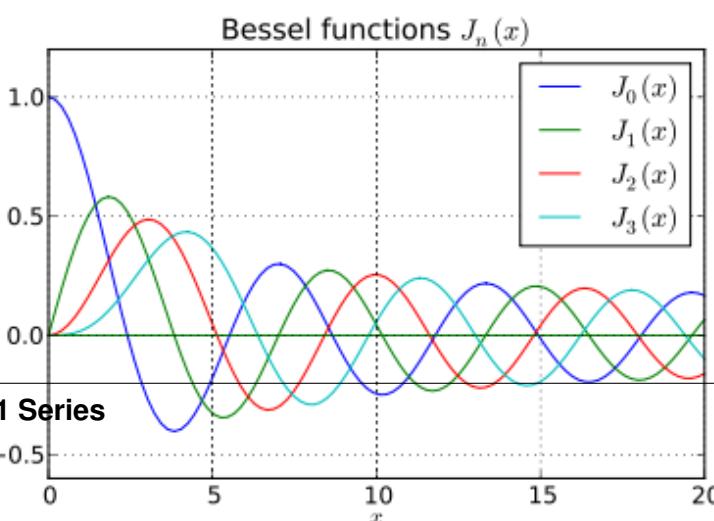


```
IPython 0.11.alpha1.git -- An enhanced Interactive Python.  
?           -> Introduction and overview of IPython's features.  
%quickref   -> Quick reference.  
help        -> Python's own help system.  
object?     -> Details about 'object', use 'object??' for extra details.  
%uiref      -> A brief reference about the graphical user interface.
```

In [1]: run recarr_simple.py



```
In [2]: from scipy import special as sp  
....: x = linspace(0, 20, 100)  
....: for n in range(4):  
....:     y = sp.jn(n, x)  
....:     plot(x, y, label=r'$J_{%s}(x)$' % n)  
....: axhline(0, color='green', label='_nolegend_')  
....: grid()  
....: legend()  
....: xlabel('$x$')  
....: title(r'Bessel functions $J_n(x)$')  
Out[2]: <matplotlib.text.Text object at 0x7fcddc1795d0>
```



Refactoring

As of this release, a significant portion of IPython 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.HasTraits`.
- `IPython.config.configurable.Configurable`.
- `IPython.config.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 biggest user-visible change is likely the move to using the config system to determine the command-line arguments for IPython applications. The benefit of this is that *all* configurable values in IPython are exposed on the command-line, but the syntax for specifying values has changed. The gist is that assigning values is pure Python assignment. Simple flags exist for commonly used options, these are always prefixed with ‘-’.

The IPython command-line help has the details of all the options (via `ipython --help`), but a simple example should clarify things; the `pylab` flag can be used to start in pylab mode with the qt4 backend:

```
ipython --pylab=qt
```

which is equivalent to using the fully qualified form:

```
ipython --TerminalIPythonApp.pylab=qt
```

The long-form options can be listed via `ipython --help-all`.

ZeroMQ architecture

There is a new GUI framework for IPython, based on a client-server model in which multiple clients can communicate with one IPython kernel, using the ZeroMQ messaging framework. There is already a Qt console client, which can be started by calling `ipython qtconsole`. The protocol is *documented*.

The parallel computing framework has also been rewritten using ZMQ. The protocol is described *here*, and the code is in the new `IPython.parallel` module.

Python 3 support

A Python 3 version of IPython has been prepared. For the time being, this is maintained separately and updated from the main codebase. Its code can be found *here*. The parallel computing components are not perfect on Python3, but most functionality appears to be working. As this work is evolving quickly, the best place to find updated information about it is our [Python 3 wiki page](#).

Unicode

Entering non-ascii characters in unicode literals (`u"\u2296"`) now works properly on all platforms. However, entering these in byte/string literals ("`\u2296`") will not work as expected on Windows (or any platform where the terminal encoding is not UTF-8, as it typically is for Linux & Mac OS X). You can use escape sequences ("`\xe9\x82`") to get bytes above 128, or use unicode literals and encode them. This is a limitation of Python 2 which we cannot easily work around.

Integration with Microsoft Visual Studio

IPython can be used as the interactive shell in the [Python plugin for Microsoft Visual Studio](#), as seen here:

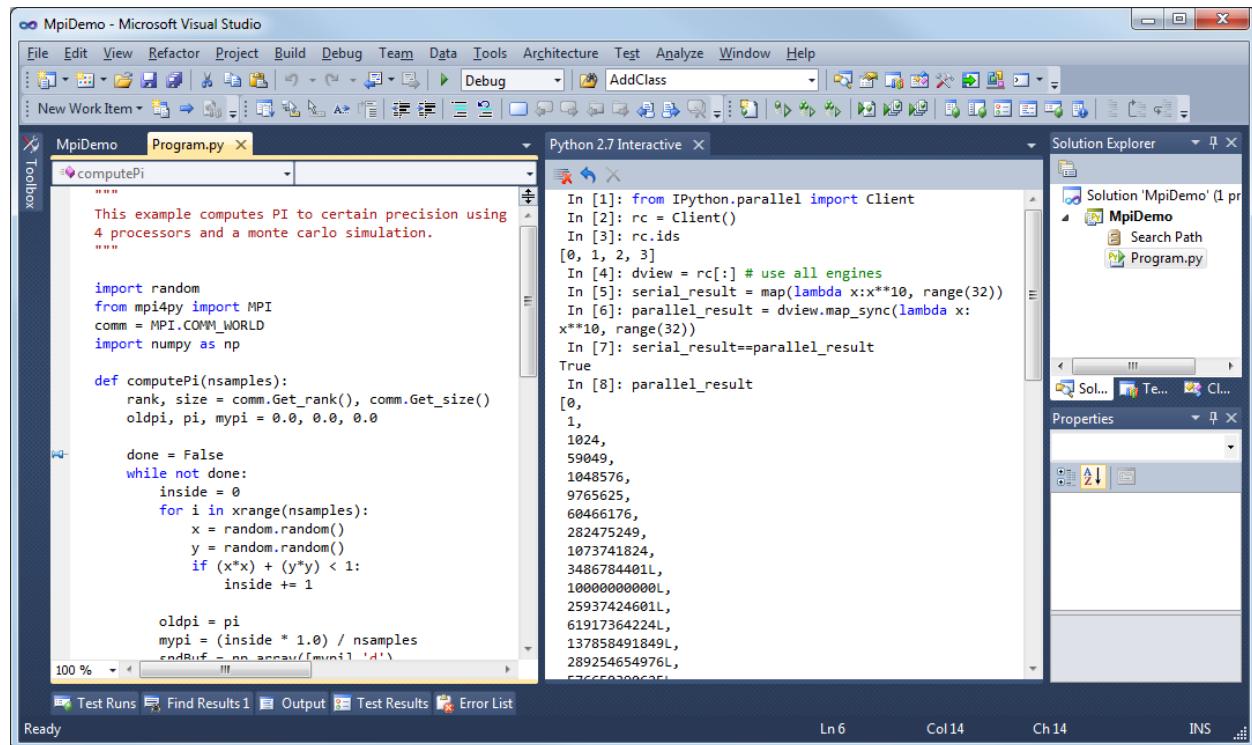


Figure 2.2: IPython console embedded in Microsoft Visual Studio.

The Microsoft team developing this currently has a release candidate out using IPython 0.11. We will continue to collaborate with them to ensure that as they approach their final release date, the integration with IPython remains smooth. We'd like to thank Dino Viehland and Shahrokh Mortazavi for the work they have done towards this feature, as well as Wenming Ye for his support of our WinHPC capabilities.

Additional new features

- Added Bytes traitlet, removing Str. All ‘string’ traitlets should either be Unicode if a real string, or Bytes if a C-string. This removes ambiguity and helps the Python 3 transition.
- New magic %loadpy loads a python file from disk or web URL into the current input buffer.

- New magic %pastebin for sharing code via the ‘Lodge it’ pastebin.
- New magic %precision for controlling float and numpy pretty printing.
- IPython applications initiate logging, so any object can gain access to the logger of the currently running Application with:

```
from IPython.config.application import Application
logger = Application.instance().log
```

- You can now get help on an object halfway through typing a command. For instance, typing a = zip? shows the details of zip(). It also leaves the command at the next prompt so you can carry on with it.
- The input history is now written to an SQLite database. The API for retrieving items from the history has also been redesigned.
- 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 Configurable 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.
- Added 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 Configurable 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).
- New top level embed() function that can be called to embed IPython at any place in user’s code. On the first call it will create an InteractiveShellEmbed instance and call it. In later calls, it just calls the previously created InteractiveShellEmbed.
- Created a configuration system (IPython.config.configurable) that is based on IPython.utils.traitlets. Configurables are arranged into a runtime containment tree (not

inheritance) that i) automatically propagates configuration information and ii) allows singletons to discover each other in a loosely coupled manner. In the future all parts of IPython will be subclasses of `Configurable`. All IPython developers should become familiar with the config 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 basic Python program. An application loads and merges all the configuration objects, constructs the main application, configures and initiates logging, and creates and configures any `Configurable` instances and then starts the application running. An extended `BaseIPythonApplication` class adds logic for handling the IPython directory as well as profiles, and all IPython entry points extend it.
- 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 Traits like implementation in `IPython.utils.traitlets`. This is a pure Python, lightweight version of a library that is similar to Enthought's Traits project, but has no dependencies on Enthought's code. We are using this for validation, defaults and notification in our new component system. Although it is not 100% API compatible with Enthought's 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.
- 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.

Backwards incompatible changes

- The Twisted-based `IPython.kernel` has been removed, and completely rewritten as `IPython.parallel`, using ZeroMQ.
- Profiles are now directories. Instead of a profile being a single config file, profiles are now self-contained directories. By default, profiles get their own IPython history, log files, and everything. To

create a new profile, do `ipython profile create <name>`.

- All IPython applications have been rewritten to use `KeyValueConfigLoader`. This means that command-line options have changed. Now, all configurable values are accessible from the command-line with the same syntax as in a configuration file.
- The command line options `-wthread`, `-qthread` and `-gthread` have been removed. Use `--gui=wx`, `--gui=qt`, `--gui=gtk` instead.
- 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 `__builtins__`.
- `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 the running IPython.
- 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. Users 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`.
- 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.
- `IPython.ipapi` has been moved to `IPython.core.ipapi`. `IPython.Shell` and `IPython.iplib` have been split and removed as part of the refactor.
- 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](#).

- Previous IPython GUIs in `IPython.frontend` and `IPython.gui` are likely broken, and have been removed to `IPython.deathrow` because of the refactoring in the core. With proper updates, these should still work.

Known Regressions

We do our best to improve IPython, but there are some known regressions in 0.11 relative to 0.10.2. First of all, there are features that have yet to be ported to the new APIs, and in order to ensure that all of the installed code runs for our users, we have moved them to two separate directories in the source distribution, *quarantine* and *deathrow*. Finally, we have some other miscellaneous regressions that we hope to fix as soon as possible. We now describe all of these in more detail.

Quarantine

These are tools and extensions that we consider relatively easy to update to the new classes and APIs, but that we simply haven't had time for. Any user who is interested in one of these is encouraged to help us by porting it and submitting a pull request on our [development site](#).

Currently, the quarantine directory contains:

| | | |
|------------------------------------|-------------------------------------|--------------------------------------|
| <code>clearcmd.py</code> | <code>ipy_fsops.py</code> | <code>ipy_signals.py</code> |
| <code>envpersist.py</code> | <code>ipy_gnuglobal.py</code> | <code>ipy_synchronize_with.py</code> |
| <code>ext_rescapture.py</code> | <code>ipy_greedycompleter.py</code> | <code>ipy_system_conf.py</code> |
| <code>InterpreterExec.py</code> | <code>ipy_jot.py</code> | <code>ipy_which.py</code> |
| <code>ipy_app_completers.py</code> | <code>ipy_lookfor.py</code> | <code>ipy_winpdb.py</code> |
| <code>ipy_autoreload.py</code> | <code>ipy_profile_doctest.py</code> | <code>ipy_workdir.py</code> |
| <code>ipy_completers.py</code> | <code>ipy_pydb.py</code> | <code>jobctrl.py</code> |
| <code>ipy_editors.py</code> | <code>ipy_rehashdir.py</code> | <code>ledit.py</code> |
| <code>ipy_exportdb.py</code> | <code>ipy_render.py</code> | <code>pspersistence.py</code> |
| <code>ipy_extutil.py</code> | <code>ipy_server.py</code> | <code>win32clip.py</code> |

Deathrow

These packages may be harder to update or make most sense as third-party libraries. Some of them are completely obsolete and have been already replaced by better functionality (we simply haven't had the time to carefully weed them out so they are kept here for now). Others simply require fixes to code that the current core team may not be familiar with. If a tool you were used to is included here, we encourage you to contact the dev list and we can discuss whether it makes sense to keep it in IPython (if it can be maintained).

Currently, the deathrow directory contains:

| | | |
|------------------------------------|-----------------------------------|---------------------------------------|
| <code>astyle.py</code> | <code>ipy_defaults.py</code> | <code>ipy_vimserver.py</code> |
| <code>dtutils.py</code> | <code>ipy_kitcfg.py</code> | <code>numeric_formats.py</code> |
| <code>Gnuplot2.py</code> | <code>ipy_legacy.py</code> | <code>numutils.py</code> |
| <code>GnuplotInteractive.py</code> | <code>ipy_p4.py</code> | <code>outputtrap.py</code> |
| <code>GnuplotRuntime.py</code> | <code>ipy_profile_none.py</code> | <code>PhysicalQIInput.py</code> |
| <code>ibrowse.py</code> | <code>ipy_profile_numpy.py</code> | <code>PhysicalQIInteractive.py</code> |
| <code>igrid.py</code> | <code>ipy_profile_scipy.py</code> | <code>quitter.py*</code> |
| <code>ipipe.py</code> | <code>ipy_profile_sh.py</code> | <code>scitedirector.py</code> |

| | | |
|------------------|-------------------------|------------|
| iplib.py | ipy_profile_zope.py | Shell.py |
| ipy_constants.py | ipy_traits_completer.py | twshell.py |

Other regressions

- The machinery that adds functionality to the ‘sh’ profile for using IPython as your system shell has not been updated to use the new APIs. As a result, only the aesthetic (prompt) changes are still implemented. We intend to fix this by 0.12. Tracked as issue [547](#).
- The installation of scripts on Windows was broken without setuptools, so we now depend on setuptools on Windows. We hope to fix setuptools-less installation, and then remove the setuptools dependency. Issue [539](#).
- The directory history `_dh` is not saved between sessions. Issue [634](#).

Removed Features

As part of the updating of IPython, we have removed a few features for the purposes of cleaning up the code-base and interfaces. These removals are permanent, but for any item listed below, equivalent functionality is available.

- The magics `Exit` and `Quit` have been dropped as ways to exit IPython. Instead, the lowercase forms of both work either as a bare name (`exit`) or a function call (`exit()`). You can assign these to other names using `exec_lines` in the config file.

Credits

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):

- Aenugu Sai Kiran Reddy <saikrn08-at-gmail.com>
- andy wilson <wilson.andrew.j+github-at-gmail.com>
- Antonio Cuni <antocuni>
- Barry Wark <barrywark-at-gmail.com>
- Beetoju Anuradha <anu.beethoju-at-gmail.com>
- Benjamin Ragan-Kelley <minrk-at-Mercury.local>
- Brad Reisfeld
- Brian E. Granger <ellisonbg-at-gmail.com>
- Christoph Gohlke <cgohlke-at-uci.edu>
- Cody Precord

- dan.milstein
- Darren Dale <dsdale24-at-gmail.com>
- Dav Clark <davclark-at-berkeley.edu>
- David Warde-Farley <wardefar-at-iro.umontreal.ca>
- epatters <ejpatters-at-gmail.com>
- epatters <epatters-at-caltech.edu>
- epatters <epatters-at-enthought.com>
- Eric Firing <efiring-at-hawaii.edu>
- Erik Tollerud <erik.tollerud-at-gmail.com>
- Evan Patterson <epatters-at-enthought.com>
- Fernando Perez <Fernando.Perez-at-berkeley.edu>
- Gael Varoquaux <gael.varoquaux-at-normalesup.org>
- Gerardo <muzgash-at-Muzpelheim>
- Jason Grout <jason.grout-at-drake.edu>
- John Hunter <jdh2358-at-gmail.com>
- Jens Hedegaard Nielsen <jenshnielsen-at-gmail.com>
- Johann Cohen-Tanugi <johann.cohentanugi-at-gmail.com>
- Jörgen Stenarson <jorgen.stenarson-at-bostream.nu>
- Justin Riley <justin.t.riley-at-gmail.com>
- Kiorky
- Laurent Dufrechou <laurient.dufrechou-at-gmail.com>
- Luis Pedro Coelho <lpc-at-cmu.edu>
- Mani chandra <mchandra-at-iitk.ac.in>
- Mark E. Smith
- Mark Voorhies <mark.voorhies-at-ucsf.edu>
- Martin Spacek <git-at-mspacek.mm.st>
- Michael Droettboom <mdroe-at-stsci.edu>
- MinRK <benjaminrk-at-gmail.com>
- muzuiget <muzuiget-at-gmail.com>
- Nick Tarleton <nick-at-quixey.com>
- Nicolas Rougier <Nicolas.rougier-at-inria.fr>
- Omar Andres Zapata Mesa <andresete.chaos-at-gmail.com>

- Paul Ivanov <pivanov314-at-gmail.com>
- Pauli Virtanen <pauli.virtanen-at-iki.fi>
- Prabhu Ramachandran
- Ramana <sramana9-at-gmail.com>
- Robert Kern <robert.kern-at-gmail.com>
- Sathesh Chandra <satheshchandra88-at-gmail.com>
- Satrajit Ghosh <satra-at-mit.edu>
- Sebastian Busch
- Skipper Seabold <jsseabold-at-gmail.com>
- Stefan van der Walt <bzr-at-mentat.za.net>
- Stephan Peijnik <debian-at-sp.or.at>
- Steven Bethard
- Thomas Kluyver <takowl-at-gmail.com>
- Thomas Spura <tomspur-at-fedoraproject.org>
- Tom Fetherston <tfetherston-at-aol.com>
- Tom MacWright
- tzanko
- vankayala sowjanya <hai.sowjanya-at-gmail.com>
- Vivian De Smedt <vds2212-at-VIVIAN>
- Ville M. Vainio <vivainio-at-gmail.com>
- Vishal Vatsa <vishal.vatsa-at-gmail.com>
- Vishnu S G <sgvishnu777-at-gmail.com>
- Walter Doerwald <walter-at-livinglogic.de>

Note: This list was generated with the output of `git log dev-0.11 HEAD --format='* %aN <%aE>' | sed 's/@/\-at\-/ | sed 's/<>//'` | sort -u after some cleanup. If you should be on this list, please add yourself.

2.3 Issues closed in the 0.11 development cycle

In this cycle, we closed a total of 511 issues, 226 pull requests and 285 regular issues; this is the full list (generated with the script `tools/github_stats.py`). We should note that a few of these were made on the 0.10.x series, but we have no automatic way of filtering the issues by branch, so this reflects all of our development over the last two years, including work already released in 0.10.2:

Pull requests (226):

- [620](#): Release notes and updates to GUI support docs for 0.11
- [642](#): fix typo in docs/examples/vim/README.rst
- [631](#): two-way vim-ipython integration
- [637](#): print is a function, this allows to properly exit ipython
- [635](#): support html representations in the notebook frontend
- [639](#): Updating the credits file
- [628](#): import pexpect from IPython.external in irunner
- [596](#): Irunner
- [598](#): Fix templates for CrashHandler
- [590](#): Desktop
- [600](#): Fix bug with non-ascii reprs inside pretty-printed lists.
- [618](#): I617
- [599](#): Gui Qt example and docs
- [619](#): manpage update
- [582](#): Updating sympy profile to match the exec_lines of isympy.
- [578](#): Check to see if correct source for decorated functions can be displayed
- [589](#): issue 588
- [591](#): simulate shell expansion on %run arguments, at least tilde expansion
- [576](#): Show message about %paste magic on an IndentationError
- [574](#): Getcwd
- [565](#): don't move old config files, keep nagging the user
- [575](#): Added more docstrings to IPython.zmq.session.
- [567](#): fix trailing whitespace from reseting indentation
- [564](#): Command line args in docs
- [560](#): reorder qt support in kernel
- [561](#): command-line suggestions
- [556](#): qt_for_kernel: use matplotlib rcParams to decide between PyQt4 and PySide
- [557](#): Update usage.py to newapp
- [555](#): Rm default old config
- [552](#): update parallel code for py3k
- [504](#): Updating string formatting

- [551](#): Make pylab import all configurable
- [496](#): Qt editing keybindings
- [550](#): Support v2 PyQt4 APIs and PySide in kernel's GUI support
- [546](#): doc update
- [548](#): Fix sympy profile to work with sympy 0.7.
- [542](#): issue 440
- [533](#): Remove unused configobj and validate libraries from externals.
- [538](#): fix various tests on Windows
- [540](#): support *-pylab* flag with deprecation warning
- [537](#): Docs update
- [536](#): *setup.py install* depends on setuptools on Windows
- [480](#): Get help mid-command
- [462](#): Str and Bytes traitlets
- [534](#): Handle unicode properly in IPython.zmq.iostream
- [527](#): ZMQ displayhook
- [526](#): Handle asynchronous output in Qt console
- [528](#): Do not import deprecated functions from external decorators library.
- [454](#): New BaseIPythonApplication
- [532](#): Zmq unicode
- [531](#): Fix Parallel test
- [525](#): fallback on lsof if otool not found in libedit detection
- [517](#): Merge IPython.parallel.streamsession into IPython.zmq.session
- [521](#): use dict.get(key) instead of dict[key] for safety from KeyErrors
- [492](#): add QtConsoleApp using newapplication
- [485](#): terminal IPython with newapp
- [486](#): Use newapp in parallel code
- [511](#): Add a new line before displaying multiline strings in the Qt console.
- [509](#): i508
- [501](#): ignore EINTR in channel loops
- [495](#): Better selection of Qt bindings when QT_API is not specified
- [498](#): Check for .pyd as extension for binary files.
- [494](#): QtConsole zoom adjustments

- [490](#): fix UnicodeEncodeError writing SVG string to .svg file, fixes #489
- [491](#): add QtConsoleApp using newapplication
- [479](#): embed() doesn't load default config
- [483](#): Links launchpad -> github
- [419](#): %xdel magic
- [477](#): Add n to lines in the log
- [459](#): use os.system for shell.system in Terminal frontend
- [475](#): i473
- [471](#): Add test decorator onlyif_unicode_paths.
- [474](#): Fix support for raw GTK and WX matplotlib backends.
- [472](#): Kernel event loop is robust against random SIGINT.
- [460](#): Share code for magic_edit
- [469](#): Add exit code when running all tests with iptest.
- [464](#): Add home directory expansion to IPYTHON_DIR environment variables.
- [455](#): Bugfix with logger
- [448](#): Separate out skip_doctest decorator
- [453](#): Draft of new main BaseIPythonApplication.
- [452](#): Use list/tuple/dict/set subclass's overridden __repr__ instead of the pretty
- [398](#): allow toggle of svg/png inline figure format
- [381](#): Support inline PNGs of matplotlib plots
- [413](#): Retries and Resubmit (#411 and #412)
- [370](#): Fixes to the display system
- [449](#): Fix issue 447 - inspecting old-style classes.
- [423](#): Allow type checking on elements of List,Tuple,Set traits
- [400](#): Config5
- [421](#): Generalise mechanism to put text at the next prompt in the Qt console.
- [443](#): pinfo code duplication
- [429](#): add check_pid, and handle stale PID info in ipcluster.
- [431](#): Fix error message in test_irunner
- [427](#): handle different SyntaxError messages in test_irunner
- [424](#): Irunner test failure
- [430](#): Small parallel doc typo

- [422](#): Make ipython-qtconsole a GUI script
- [420](#): Permit kernel std* to be redirected
- [408](#): History request
- [388](#): Add Emacs-style kill ring to Qt console
- [414](#): Warn on old config files
- [415](#): Prevent prefilter from crashing IPython
- [418](#): Minor configuration doc fixes
- [407](#): Update What's new documentation
- [410](#): Install notebook frontend
- [406](#): install IPython.zmq.gui
- [393](#): ipdir unicode
- [397](#): utils.io.Term.cin/out/err -> utils.io.stdin/out/err
- [389](#): DB fixes and Scheduler HWM
- [374](#): Various Windows-related fixes to IPython.parallel
- [362](#): fallback on defaultencoding if filesystemencoding is None
- [382](#): Shell's reset method clears namespace from last %run command.
- [385](#): Update iptest exclusions (fix #375)
- [383](#): Catch errors in querying readline which occur with pyreadline.
- [373](#): Remove runlines etc.
- [364](#): Single output
- [372](#): Multiline input push
- [363](#): Issue 125
- [361](#): don't rely on setuptools for readline dependency check
- [349](#): Fix %autopx magic
- [355](#): History save thread
- [356](#): Usability improvements to history in Qt console
- [357](#): Exit autocall
- [353](#): Rewrite quit()/exit()/Quit()/Exit() calls as magic
- [354](#): Cell tweaks
- [345](#): Attempt to address (partly) issue ipython/#342 by rewriting quit(), exit(), etc.
- [352](#): #342: Try to recover as intelligently as possible if user calls magic().
- [346](#): Dedent prefix bugfix + tests: #142

- [348](#): %reset doesn't reset prompt number.
- [347](#): Make ip.reset() work the same in interactive or non-interactive code.
- [343](#): make readline a dependency on OSX
- [344](#): restore auto debug behavior
- [339](#): fix for issue 337: incorrect/phantom tooltips for magics
- [254](#): newparallel branch (add zmq.parallel submodule)
- [334](#): Hard reset
- [316](#): Unicode win process
- [332](#): AST splitter
- [325](#): Removetwisted
- [330](#): Magic pastebin
- [309](#): Bug tests for GH Issues 238, 284, 306, 307. Skip module machinery if not installed. Known failures reported as 'K'
- [331](#): Tweak config loader for PyPy compatibility.
- [319](#): Rewrite code to restore readline history after an action
- [329](#): Do not store file contents in history when running a .ipy file.
- [179](#): Html notebook
- [323](#): Add missing external.pexpect to packages
- [295](#): Magic local scope
- [315](#): Unicode magic args
- [310](#): allow Unicode Command-Line options
- [313](#): Readline shortcuts
- [311](#): Qtconsole exit
- [312](#): History memory
- [294](#): Issue 290
- [292](#): Issue 31
- [252](#): Unicode issues
- [235](#): Fix history magic command's bugs wrt to full history and add -O option to display full history
- [236](#): History minus p flag
- [261](#): Adapt magic commands to new history system.
- [282](#): SQLite history
- [191](#): Unbundle external libraries

- [199](#): Magic arguments
- [204](#): Emacs completion bugfix
- [293](#): Issue 133
- [249](#): Writing unicode characters to a log file. (IPython 0.10.2.git)
- [283](#): Support for 256-color escape sequences in Qt console
- [281](#): Refactored and improved Qt console's HTML export facility
- [237](#): Fix185 (take two)
- [251](#): Issue 129
- [278](#): add basic XDG_CONFIG_HOME support
- [275](#): inline pylab cuts off labels on log plots
- [280](#): Add %precision magic
- [259](#): Pyside support
- [193](#): Make ipython cProfile-able
- [272](#): Magic examples
- [219](#): Doc magic pycat
- [221](#): Doc magic alias
- [230](#): Doc magic edit
- [224](#): Doc magic cpaste
- [229](#): Doc magic pdef
- [273](#): Docs build
- [228](#): Doc magic who
- [233](#): Doc magic cd
- [226](#): Doc magic pwd
- [218](#): Doc magic history
- [231](#): Doc magic reset
- [225](#): Doc magic save
- [222](#): Doc magic timeit
- [223](#): Doc magic colors
- [203](#): Small typos in zmq/blockingkernelmanager.py
- [227](#): Doc magic logon
- [232](#): Doc magic profile
- [264](#): Kernel logging

- [220](#): Doc magic edit
- [268](#): PyZMQ >= 2.0.10
- [267](#): GitHub Pages (again)
- [266](#): OSX-specific fixes to the Qt console
- [255](#): Gitwash typo
- [265](#): Fix string input2
- [260](#): Kernel crash with empty history
- [243](#): New display system
- [242](#): Fix terminal exit
- [250](#): always use Session.send
- [239](#): Makefile command & script for GitHub Pages
- [244](#): My exit
- [234](#): Timed history save
- [217](#): Doc magic lsmagic
- [215](#): History fix
- [195](#): Formatters
- [192](#): Ready colorize bug
- [198](#): Windows workdir
- [174](#): Whitespace cleanup
- [188](#): Version info: update our version management system to use git.
- [158](#): Ready for merge
- [187](#): Resolved Print shortcut collision with ctrl-P emacs binding
- [183](#): cleanup of exit/quit commands for qt console
- [184](#): Logo added to sphinx docs
- [180](#): Cleanup old code
- [171](#): Expose Pygments styles as options
- [170](#): HTML Fixes
- [172](#): Fix del method exit test
- [164](#): Qt frontend shutdown behavior fixes and enhancements
- [167](#): Added HTML export
- [163](#): Execution refactor
- [159](#): Ipy3 preparation

- [155](#): Ready startup fix
- [152](#): 0.10.1 sge
- [151](#): mk_object_info -> object_info
- [149](#): Simple bug-fix

Regular issues (285):

- [630](#): new.py in pwd prevents ipython from starting
- [623](#): Execute DirectView commands while running LoadBalancedView tasks
- [437](#): Users should have autocompletion in the notebook
- [583](#): update manpages
- [594](#): irunner command line options defer to file extensions
- [603](#): Users should see colored text in tracebacks and the pager
- [597](#): UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xc2
- [608](#): Organize and layout buttons in the notebook panel sections
- [609](#): Implement controls in the Kernel panel section
- [611](#): Add kernel status widget back to notebook
- [610](#): Implement controls in the Cell section panel
- [612](#): Implement Help panel section
- [621](#): [qtconsole] on windows xp, cannot PageUp more than once
- [616](#): Store exit status of last command
- [605](#): Users should be able to open different notebooks in the cwd
- [302](#): Users should see a consistent behavior in the Out prompt in the html notebook
- [435](#): Notebook should not import anything by default
- [595](#): qtconsole command issue
- [588](#): ipython-qtconsole uses 100% CPU
- [586](#): ? + plot() Command B0rks QTConsole Strangely
- [585](#): %pdoc throws Errors for classes without __init__ or docstring
- [584](#): %pdoc throws TypeError
- [580](#): Client instantiation AssertionException
- [569](#): UnicodeDecodeError during startup
- [572](#): Indented command hits error
- [573](#): -wthread breaks indented top-level statements
- [570](#): “–pylab inline” vs. “–pylab=inline”

- [566](#): Can't use exec_file in config file
- [562](#): update docs to reflect ‘–args=values’
- [558](#): triple quote and %s at beginning of line
- [554](#): Update 0.11 docs to explain Qt console and how to do a clean install
- [553](#): embed() fails if config files not installed
- [8](#): Ensure %gui qt works with new Mayavi and pylab
- [269](#): Provide compatibility api for IPython.Shell().start().mainloop()
- [66](#): Update the main What's New document to reflect work on 0.11
- [549](#): Don't check for 'linux2' value in sys.platform
- [505](#): Qt windows created within imported functions won't show()
- [545](#): qtconsole ignores exec_lines
- [371](#): segfault in qtconsole when kernel quits
- [377](#): Failure: error (nothing to repeat)
- [544](#): Ipython qtconsole pylab config issue.
- [543](#): RuntimeError in completer
- [440](#): %run filename autocompletion “The kernel heartbeat has been inactive ... ” error
- [541](#): log_level is broken in the ipython Application
- [369](#): windows source install doesn't create scripts correctly
- [351](#): Make sure that the Windows installer handles the top-level IPython scripts.
- [512](#): Two displayhooks in zmq
- [340](#): Make sure that the Windows HPC scheduler support is working for 0.11
- [98](#): Should be able to get help on an object mid-command
- [529](#): unicode problem in qtconsole for windows
- [476](#): Separate input area in Qt Console
- [175](#): Qt console needs configuration support
- [156](#): Key history lost when debugging program crash
- [470](#): decorator: uses deprecated features
- [30](#): readline in OS X does not have correct key bindings
- [503](#): merge IPython.parallel.streamsession and IPython.zmq.session
- [456](#): pathname in document punctuated by dots not slashes
- [451](#): Allow switching the default image format for inline mpl backend
- [79](#): Implement more robust handling of config stages in Application

- [522](#): Encoding problems
- [524](#): otool should not be unconditionally called on osx
- [523](#): Get profile and config file inheritance working
- [519](#): qtconsole –pure: “TypeError: string indices must be integers, not str”
- [516](#): qtconsole –pure: “KeyError: ‘ismagic’”
- [520](#): qtconsole –pure: “TypeError: string indices must be integers, not str”
- [450](#): resubmitted tasks sometimes stuck as pending
- [518](#): JSON serialization problems with ObjectId type (MongoDB)
- [178](#): Channels should be named for their function, not their socket type
- [515](#): [ipcluster] termination on os x
- [510](#): qtconsole: indentation problem printing numpy arrays
- [508](#): “AssertionError: Missing message part.” in ipython-qtconsole –pure
- [499](#): “ZMQError: Interrupted system call” when saving inline figure
- [426](#): %edit magic fails in qtconsole
- [497](#): Don’t show info from .pyd files
- [493](#): QFont::setPointSize: Point size <= 0 (0), must be greater than 0
- [489](#): UnicodeEncodeError in qt.svg.save_svg
- [458](#): embed() doesn’t load default config
- [488](#): Using IPython with RubyPython leads to problems with IPython.parallel.client.Client.__init__()
- [401](#): Race condition when running lbview.apply() fast multiple times in loop
- [168](#): Scrub Launchpad links from code, docs
- [141](#): garbage collection problem (revisited)
- [59](#): test_magic.test_obj_del fails on win32
- [457](#): Backgrounded Tasks not Allowed? (but easy to slip by . . .)
- [297](#): Shouldn’t use pexpect for subprocesses in in-process terminal frontend
- [110](#): magic to return exit status
- [473](#): OSX readline detection fails in the debugger
- [466](#): tests fail without unicode filename support
- [468](#): iptest script has 0 exit code even when tests fail
- [465](#): client.db_query() behaves different with SQLite and MongoDB
- [467](#): magic_install_default_config test fails when there is no .ipython directory

- [463](#): IPYTHON_DIR (and IPYTHONDIR) don't expand tilde to ‘~’ directory
- [446](#): Test machinery is imported at normal runtime
- [438](#): Users should be able to use Up/Down for cell navigation
- [439](#): Users should be able to copy notebook input and output
- [291](#): Rename special display methods and put them lower in priority than display functions
- [447](#): Instantiating classes without `__init__` function causes kernel to crash
- [444](#): Ctrl + t in WxIPython Causes Unexpected Behavior
- [445](#): qt and console Based Startup Errors
- [428](#): ipcluster doesn't handle stale pid info well
- [434](#): 10.0.2 seg fault with rpy2
- [441](#): Allow running a block of code in a file
- [432](#): Silent request fails
- [409](#): Test failure in IPython.lib
- [402](#): History section of messaging spec is incorrect
- [88](#): Error when inputting UTF8 CJK characters
- [366](#): Ctrl-K should kill line and store it, so that Ctrl-y can yank it back
- [425](#): typo in %gui magic help
- [304](#): Persistent warnings if old configuration files exist
- [216](#): crash of ipython when alias is used with %s and echo
- [412](#): add support to automatic retry of tasks
- [411](#): add support to continue tasks
- [417](#): IPython should display things unsorted if it can't sort them
- [416](#): wrong encode when printing unicode string
- [376](#): Failing InputsplitterTest
- [405](#): TraitError in traitlets.py(332) on any input
- [392](#): UnicodeEncodeError on start
- [137](#): sys.getfilesystemencoding return value not checked
- [300](#): Users should be able to manage kernels and kernel sessions from the notebook UI
- [301](#): Users should have access to working Kernel, Tabs, Edit, Help menus in the notebook
- [396](#): cursor move triggers a lot of IO access
- [379](#): Minor doc nit: –paging argument
- [399](#): Add task queue limit in engine when load-balancing

- [78](#): StringTask won't take unicode code strings
- [391](#): MongoDB.add_record() does not work in 0.11dev
- [365](#): newparallel on Windows
- [386](#): FAIL: test that pushed functions have access to globals
- [387](#): Interactively defined functions can't access user namespace
- [118](#): Snow Leopard ipy_vimserver POLL error
- [394](#): System escape interpreted in multi-line string
- [26](#): find_job_cmd is too hasty to fail on Windows
- [368](#): Installation instructions in dev docs are completely wrong
- [380](#): qtconsole pager RST - HTML not happening consistently
- [367](#): Qt console doesn't support ibus input method
- [375](#): Missing libraries cause ImportError in tests
- [71](#): temp file errors in iptest IPython.core
- [350](#): Decide how to handle displayhook being triggered multiple times
- [360](#): Remove *runlines* method
- [125](#): Exec lines in config should not contribute to line numbering or history
- [20](#): Robust readline support on OS X's builtin Python
- [147](#): On Windows, %page is being too restrictive to split line by rn only
- [326](#): Update docs and examples for parallel stuff to reflect movement away from Twisted
- [341](#): Fix Parallel Magics for newparallel
- [338](#): Usability improvements to Qt console
- [142](#): unexpected auto-indenting when variables names that start with 'pass'
- [296](#): Automatic PDB via %pdb doesn't work
- [337](#): exit(and quit(in Qt console produces phantom signature/docstring popup, even though quit() or exit() raises NameError
- [318](#): %debug broken in master: invokes missing save_history() method
- [307](#): lines ending with semicolon should not go to cache
- [104](#): have ipengine run start-up scripts before registering with the controller
- [33](#): The skip_doctest decorator is failing to work on Shell.MatplotlibShellBase.magic_run
- [336](#): Missing figure development/figs/iopubfade.png for docs
- [49](#): %clear should also delete _NN references and Out[NN] ones
- [335](#): using setuptools installs every script twice

- [306](#): multiline strings at end of input cause noop
- [327](#): PyPy compatibility
- [328](#): %run script.ipynb raises “ERROR! Session/line number was not unique in database.”
- [7](#): Update the changes doc to reflect the kernel config work
- [303](#): Users should be able to scroll a notebook w/o moving the menu/buttons
- [322](#): Embedding an interactive IPython shell
- [321](#): %debug broken in master
- [287](#): Crash when using %macros in sqlite-history branch
- [55](#): Can’t edit files whose names begin with numbers
- [284](#): In variable no longer works in 0.11
- [92](#): Using multiprocessing module crashes parallel iPython
- [262](#): Fail to recover history after force-kill.
- [320](#): Tab completing re.search objects crashes IPython
- [317](#): IPython.kernel: parallel map issues
- [197](#): ipython-qtconsole unicode problem in magic ls
- [305](#): more readline shortcuts in qtconsole
- [314](#): Multi-line, multi-block cells can’t be executed.
- [308](#): Test suite should set sqlite history to work in :memory:
- [202](#): Matplotlib native ‘MacOSX’ backend broken in ‘-pylab’ mode
- [196](#): IPython can’t deal with unicode file name.
- [25](#): unicode bug - encoding input
- [290](#): try/except/else clauses can’t be typed, code input stops too early.
- [43](#): Implement SSH support in ipcluster
- [6](#): Update the Sphinx docs for the new ipcluster
- [9](#): Getting “DeadReferenceError: Calling Stale Broker” after ipcontroller restart
- [132](#): Ipython prevent south from working
- [27](#): generics.complete_object broken
- [60](#): Improve absolute import management for iptest.py
- [31](#): Issues in magic_whos code
- [52](#): Document testing process better
- [44](#): Merge history from multiple sessions
- [182](#): ipython q4thread in version 10.1 not starting properly

- [143](#): Ipython.gui.wx.ipython_view.IPShellWidget: ignores user*_ns arguments
- [127](#): %edit does not work on filenames consisted of pure numbers
- [126](#): Can't transfer command line argument to script
- [28](#): Offer finer control for initialization of input streams
- [58](#): ipython change char '0xe9' to 4 spaces
- [68](#): Problems with Control-C stopping ipcluster on Windows/Python2.6
- [24](#): ipcluster does not start all the engines
- [240](#): Incorrect method displayed in %psource
- [120](#): inspect.getsource fails for functions defined on command line
- [212](#): IPython ignores exceptions in the first evaluation of class attrs
- [108](#): ipython disables python logger
- [100](#): Overzealous introspection
- [18](#): %cpaste freeze sync frontend
- [200](#): Unicode error when starting ipython in a folder with non-ascii path
- [130](#): Deadlock when importing a module that creates an IPython client
- [134](#): multiline block scrolling
- [46](#): Input to %timeit is not prepared
- [285](#): ipcluster local -n 4 fails
- [205](#): In the Qt console, Tab should insert 4 spaces when not completing
- [145](#): Bug on MSW systems: idle can not be set as default IPython editor. Fix Suggested.
- [77](#): ipython oops in cygwin
- [121](#): If plot windows are closed via window controls, no more plotting is possible.
- [111](#): Iterator version of TaskClient.map() that returns results as they become available
- [109](#): WinHPCLauncher is a hard dependency that causes errors in the test suite
- [86](#): Make IPython work with multiprocessing
- [15](#): Implement SGE support in ipcluster
- [3](#): Implement PBS support in ipcluster
- [53](#): Internal Python error in the inspect module
- [74](#): Manager() [from multiprocessing module] hangs ipythonx but not ipython
- [51](#): Out not working with ipythonx
- [201](#): use session.send throughout zmq code
- [115](#): multiline specials not defined in 0.11 branch

- [93](#): when looping, cursor appears at leftmost point in newline
- [133](#): whitespace after Source introspection
- [50](#): Ctrl-C with -gthread on Windows, causes uncaught IOError
- [65](#): Do not use .message attributes in exceptions, deprecated in 2.6
- [76](#): syntax error when raise is inside except process
- [107](#): bdist_rpm causes traceback looking for a non-existant file
- [113](#): initial magic ? (question mark) fails before wildcard
- [128](#): Pdb instance has no attribute ‘curframe’
- [139](#): running with -pylab pollutes namespace
- [140](#): malloc error during tab completion of numpy array member functions starting with ‘c’
- [153](#): ipy_vimserver traceback on Windows
- [154](#): using ipython in Slicer3 show how os.environ['HOME'] is not defined
- [185](#): show() blocks in pylab mode with ipython 0.10.1
- [189](#): Crash on tab completion
- [274](#): bashism in sshx.sh
- [276](#): Calling *sip.setapi* does not work if app has already imported from PyQt4
- [277](#): matplotlib.image imgshow from 10.1 segfault
- [288](#): Incorrect docstring in zmq/kernelmanager.py
- [286](#): Fix IPython.Shell compatibility layer
- [99](#): blank lines in history
- [129](#): psearch: TypeError: expected string or buffer
- [190](#): Add option to format float point output
- [246](#): Application not conforms XDG Base Directory Specification
- [48](#): IPython should follow the XDG Base Directory spec for configuration
- [176](#): Make client-side history persistence readline-independent
- [279](#): Backtraces when using ipdb do not respect -colour LightBG setting
- [119](#): Broken type filter in magic_who_ls
- [271](#): Intermittent problem with print output in Qt console.
- [270](#): Small typo in IPython developer’s guide
- [166](#): Add keyboard accelerators to Qt close dialog
- [173](#): asymmetrical ctrl-A/ctrl-E behavior in multiline
- [45](#): Autosave history for robustness

- [162](#): make command history persist in ipythonqt
- [161](#): make ipythonqt exit without dialog when exit() is called
- [263](#): [ipython + numpy] Some test errors
- [256](#): reset docstring ipython 0.10
- [258](#): allow caching to avoid matplotlib object references
- [248](#): Can't open and read files after upgrade from 0.10 to 0.10.0
- [247](#): ipython + Stackless
- [245](#): Magic save and macro missing newlines, line ranges don't match prompt numbers.
- [241](#): "exit" hangs on terminal version of IPython
- [213](#): ipython -pylab no longer plots interactively on 0.10.1
- [4](#): wx frontend don't display well commands output
- [5](#): ls command not supported in ipythonx wx frontend
- [1](#): Document winhpcjob.py and launcher.py
- [83](#): Usage of testing.util.DeferredTestCase should be replace with twisted.trial.unittest.TestCase
- [117](#): Redesign how Component instances are tracked and queried
- [47](#): IPython.kernel.client cannot be imported inside an engine
- [105](#): Refactor the task dependencies system
- [210](#): 0.10.1 doc mistake - New IPython Sphinx directive error
- [209](#): can't activate IPython parallel magics
- [206](#): Buggy linewrap in Mac OSX Terminal
- [194](#): !sudo <command> displays password in plain text
- [186](#): %edit issue under OS X 10.5 - IPython 0.10.1
- [11](#): Create a daily build PPA for ipython
- [144](#): logo missing from sphinx docs
- [181](#): cls command does not work on windows
- [169](#): Kernel can only be bound to localhost
- [36](#): tab completion does not escape ()
- [177](#): Report tracebacks of interactively entered input
- [148](#): dictionary having multiple keys having frozenset fails to print on iPython
- [160](#): magic_gui throws TypeError when gui magic is used
- [150](#): History entries ending with parentheses corrupt command line on OS X 10.6.4
- [146](#): -ipythondir - using an alternative .ipython dir for rc type stuff

- [114](#): Interactive strings get mangled with “_jp.magic”
- [135](#): crash on invalid print
- [69](#): Usage of “mycluster” profile in docs and examples
- [37](#): Fix colors in output of ResultList on Windows

2.4 0.10 series

2.4.1 Release 0.10.2

IPython 0.10.2 was released April 9, 2011. This is a minor bugfix release that preserves backward compatibility. At this point, all IPython development resources are focused on the 0.11 series that includes a complete architectural restructuring of the project as well as many new capabilities, so this is likely to be the last release of the 0.10.x series. We have tried to fix all major bugs in this series so that it remains a viable platform for those not ready yet to transition to the 0.11 and newer codebase (since that will require some porting effort, as a number of APIs have changed).

Thus, we are not opening a 0.10.3 active development branch yet, but if the user community requires new patches and is willing to maintain/release such a branch, we’ll be happy to host it on the IPython github repositories.

Highlights of this release:

- The main one is the closing of github ticket #185, a major regression we had in 0.10.1 where pylab mode with GTK (or gthread) was not working correctly, hence plots were blocking with GTK. Since this is the default matplotlib backend on Unix systems, this was a major annoyance for many users. Many thanks to Paul Ivanov for helping resolve this issue.
- Fix IOError bug on Windows when used with -gthread.
- Work robustly if \$HOME is missing from environment.
- Better POSIX support in ssh scripts (remove bash-specific idioms).
- Improved support for non-ascii characters in log files.
- Work correctly in environments where GTK can be imported but not started (such as a linux text console without X11).

For this release we merged 24 commits, contributed by the following people (please let us know if we omitted your name and we’ll gladly fix this in the notes for the future):

- Fernando Perez
- MinRK
- Paul Ivanov
- Pieter Cristiaan de Groot
- TvrtnkoM

2.4.2 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]: [
```

- 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

2.4.3 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 Dufréchou: 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.demo` 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.demo` 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 Dufréchou. 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 bzr 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)

2.5 0.9 series

2.5.1 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.

2.5.2 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.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.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 mulitple variable follows that of *map*, iii) both the multiengine and task controller now have a parallel function implementation.
 - All of the parallel computing capabilities from *ipython1-dev* have been merged into IPython proper. This resulted in the following new subpackages: `IPython.kernel`, `IPython.kernel.core`, `IPython.config`, `IPython.tools` and `IPython.testing`.
 - As part of merging in the *ipython1-dev* stuff, the `setup.py` script and friends have been completely refactored. Now we are checking for dependencies using the approach that matplotlib uses.
 - The documentation has been completely reorganized to accept the documentation from *ipython1-dev*.
 - We have switched to using Foolscap for all of our network protocols in `IPython.kernel`. This gives us secure connections that are both encrypted and authenticated.
 - We have a brand new *COPYING.txt* files that describes the IPython license and copyright. The biggest change is that we are putting “The IPython Development Team” as the copyright holder. We give more details about exactly what this means in this file. All developer should read this and use the new banner in all IPython source code files.
- sh profile: `./foo` runs foo as system command, no need to do `!./foo` anymore
 - String lists now support `sort(field, nums = True)` method (to easily sort system command output). Try it with `a = !ls -l ; a.sort(1, nums=1)`.
 - ‘%cpaste foo’ now assigns the pasted block as string list, instead of string

- The ipcluster script now run by default with no security. This is done because the main usage of the script is for starting things on localhost. Eventually when ipcluster is able to start things on other hosts, we will put security back.
- ‘cd –foo’ searches directory history for string foo, and jumps to that dir. Last part of dir name is checked first. If no matches for that are found, look at the whole path.

Bug fixes

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

Backwards incompatible changes

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

- The command line options to *ipcontroller* and *ipengine* have changed to reflect the new Foolscap 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 Foolscap 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 `setupegg.py` scripts. Because Twisted and `zope.interface` are now easy installable, we can declare them as dependencies in our `setupegg.py` script.
- IPython is now compatible with Twisted 2.5.0 and 8.x.
- Added a new example of how to use `ipython1.kernel.asynclient`.
- 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 simple 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.

2.6 0.8 series

2.6.1 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.

2.6.2 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.

2.6.3 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.

2.6.4 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 <= 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.

Note: On Windows, IPython *does* depend on `setuptools`, and it is recommended that you install the `distribute` package, which improves `setuptools` and fixes various bugs.

We hope to remove this dependency in 0.12.

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 `pymq`, 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.

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. If you install IPython with `setuptools`, (e.g. with `easy_install`), then the appropriate `readline` for your platform will be installed. 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 simple 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

Note: On Windows, IPython requires `setuptools` or `distribute`.

We hope to remove this dependency in 0.12.

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 [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 [Git](#) source code repository. To do this you will need to have Git installed on your system. Then just do:

```
$ git clone https://github.com/ipython/ipython.git  
$ cd ipython  
$ python setup.py install
```

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

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

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

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

```
$ git 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.

When IPython is installed with `setuptools`, (e.g. with `easy_install`), `readline` is added as a dependency on OS X, and `PyReadline` on Windows, and will be installed on your system. However, if you do not use `setuptools`, you may have to install one of these packages yourself.

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](#).

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](#).

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, as well as for managing subprocesses [pexpect]. IPython now includes a version of pexpect in `IPython.external`, but if you have installed pexpect, IPython will use that instead. 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. If you are on Python 2.6 or 2.7 on OSX, or 2.7 on Windows, `pyzmq` has eggs that include ZeroMQ itself.

`IPython.zmq` depends on `pyzmq >= 2.1.4`.

3.7 Dependencies for ipython qtconsole (new GUI)

3.7.1 Qt

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 Qt, and works with either PyQt, which can be installed from the [PyQt website](#), or [PySide](#), from Nokia.

3.7.2 pygments

The syntax-highlighting in ipython qtconsole is done with the pygments project, which is easy_install-able [[pygments](#)].

USING IPYTHON FOR INTERACTIVE WORK

4.1 Introducing IPython

You don't need to know anything beyond Python to start using IPython – just type commands as you would at the standard Python prompt. But IPython can do much more than the standard prompt. Some key features are described here. For more information, check the [tips page](#), or look at examples in the [IPython cookbook](#).

If you've never used Python before, you might want to look at the [official tutorial](#) or an alternative, [Dive into Python](#).

4.1.1 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>` to view the object's attributes (see [the readline section](#) for more). Besides Python objects and keywords, tab completion also works on file and directory names.

4.1.2 Exploring 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. To get specific information on an object, you can use the magic commands `%pdoc`, `%pdef`, `%psource` and `%pfile`

4.1.3 Magic functions

IPython has a set of predefined ‘magic functions’ that you can call with a command line style syntax. These include:

- Functions that work with code: `%run`, `%edit`, `%save`, `%macro`, `%recall`, etc.
- Functions which affect the shell: `%colors`, `%xmode`, `%autoindent`, etc.
- Other functions such as `%reset`, `%timeit` or `%paste`.

You can always call these using the % prefix, and if you're typing one on a line by itself, you can omit even that:

```
run thescript.py
```

For more details on any magic function, call %*somemagic?* to read its docstring. To see all the available magic functions, call %*lsmagic*.

Running and Editing

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 (unlike imported modules, which have to be specifically reloaded). IPython also includes *dreload*, a recursive reload function.

%run has special flags for timing the execution of your scripts (-t), or for running them under the control of either Python's pdb debugger (-d) or profiler (-p).

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.

Debugging

After an exception occurs, you can call %debug to jump into the Python debugger (pdb) and examine the problem. Alternatively, if you call %pdb, IPython will automatically start the debugger on any uncaught exception. 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. Running programs with %run and pdb active can be an efficient way to develop and debug code, in many cases eliminating the need for print statements or external debugging tools.

You can also step through a program from the beginning by calling %run -d theprogram.py.

4.1.4 History

IPython stores both the commands you enter, and the results it produces. You can easily go through previous commands with the up- and down-arrow keys, or access your history in more sophisticated ways.

Input and output history are kept in variables called In and Out, which can both be indexed by the prompt number on which they occurred, e.g. In[4]. The last three objects in output history are also kept in variables named __, ___ and ____.

You can use the %history magic function to examine past input and output. Input history from previous sessions is saved in a database, and IPython can be configured to save output history.

Several other magic functions can use your input history, including %edit, %rerun, %recall, %macro, %save and %pastebin. You can use a standard format to refer to lines:

```
%pastebin 3 18-20 ~1/1-5
```

This will take line 3 and lines 18 to 20 from the current session, and lines 1-5 from the previous session.

4.1.5 System shell commands

To run any command at the system shell, simply prefix it with !, e.g.:

```
!ping www.bbc.co.uk
```

You can capture the output into a Python list, e.g.: `files = !ls`. To pass the values of Python variables or expressions to system commands, prefix them with \$: `!grep -rf $pattern ipython/*`. See [our shell section](#) for more details.

Define your own system aliases

It's 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.

4.1.6 Configuration

Much of IPython can be tweaked through configuration. To get started, use the command `ipython profile create` to produce the default config files. These will be placed in `~/.ipython/profile_default` or `~/.config/ipython/profile_default`, and contain comments explaining what the various options do.

Profiles allow you to use IPython for different tasks, keeping separate config files and history for each one. More details in [the profiles section](#).

4.2 IPython Tips & Tricks

The IPython cookbook details more things you can do with IPython.

4.2.1 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 [the embedding section](#).

4.2.2 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 -t` call to see your translated history allows for an easy doctest workflow, where you can go from doctest to interactive execution to pasting into valid Python code as needed.

4.2.3 Use IPython to present interactive demos

Use the `IPython.lib.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.

4.2.4 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.

4.2.5 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-zA.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.

4.3 IPython reference

4.3.1 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 `ipython_config.py`. 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.

Eventloop integration

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 removed. Please see the new `%gui` magic command or [this section](#) for details on the new interface, or specify the `gui` at the commandline:

```
$ ipython --gui=qt
```

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 (`-no-option` instead of `-option`) to turn the feature off.

`-h, --help` print a help message and exit.

`--pylab, pylab=<name>` 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=<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` 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).

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

--logfile=<name> specify the name of your logfile.

This implies `%logstart` at the beginning of your session

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 with `ipython --i ipython_log.py`

--logplay=<name>

NOT AVAILABLE in 0.11

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.

--[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=<name>

Select the IPython profile by name.

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/profile_default/ipython_config.py` file and then have other ‘profiles’ which include this one and load extra things for particular tasks. For example:

1. `$IPYTHON_DIR/profile_default` : load basic things you always want.
2. `$IPYTHON_DIR/profile_math` : load (1) and basic math-related modules.
3. `$IPYTHON_DIR/profile_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.

InteractiveShell.prompt_in1=<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.

InteractiveShell.prompt_in2=<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 [#]’).

InteractiveShell.prompt_out=<string> String used for output prompts, also uses numbers like `prompt_in1`. Default: ‘Out[#]’

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

config_file=<name> name of your IPython resource configuration file. Normally IPython loads `ipython_config.py` (from current directory) or `IPYTHON_DIR/profile_default`.

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.

--TerminalInteractiveShell.screen_length=<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.

```
--TerminalInteractiveShell.separate_in=<string>
    separator before input prompts. Default: 'n'
--TerminalInteractiveShell.separate_out=<string> separator before output prompts. Default: nothing.
--TerminalInteractiveShell.separate_out2=<string> separator after output prompts. Default: nothing. For these three options, use the value 0 to specify no separator.
--nosep shorthand for setting the above separators to empty strings.

    Simply removes all input/output separators.

--init allows you to initialize a profile dir for configuration when you install a new version of IPython or want to use a new profile. Since new versions may include new command line options or example files, this copies updated config files. Note that you should probably use %upgrade instead, it's a safer alternative.

--version print version information and exit.

--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).
```

4.3.2 Interactive use

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 changes your working directory to ‘mydir’, if it exists.

If you have ‘automagic’ enabled (as it by default), 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:

```
ip = get_ipython()

def doimp(self, arg):
    ip = self.api

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

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

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 below `<dynamic_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.core.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/profile_<name>/history.sqlite. 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: Setting the above indents will cause problems with unicode text entry in the terminal.

Warning: Autoindent 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 `ipython_config.py` file (set `TerminalInteractiveShell.autoindent=False`).

If you want to paste multiple lines, it is recommended that you use `%paste`.

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 switch `--logfile=foo.py` (see [here](#)) or by activating the logging at any moment with the magic function `%logstart`.

Log files can later be reloaded by running them as scripts 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 ‘ipython_log.py’ in your current working 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 %rehashx magic allows you to load your entire \$PATH as ipython aliases. See its docstring 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 cgitb 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, _ii, _iii: store previous, next previous and next-next previous inputs.
- In, _ih : a list of all inputs; _ih[n] is the input from line n. 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), so `_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).

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? 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 is handy for searching for URLs, IP addresses, etc. You can bring history entries listed by '`%hist -g`' up for editing with the `%recall` command, or run them immediately with `%rerun`.

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
```

4.3.3 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:

```
from IPython.frontend.terminal.ipapp import launch_new_instance
launch_new_instance()
raise SystemExit
```

then IPython will be your working environment anytime you start Python. The `raise SystemExit` is needed to exit Python 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.

4.3.4 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 import embed

embed() # 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 `embed` 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)
from IPython.config.loader import Config
try:
    get_ipython
except NameError:
    nested = 0
    cfg = Config()
    shell_config = cfg.InteractiveShellEmbed
    shell_config.prompt_in1 = 'In <\\#>: '
    shell_config.prompt_in2 = '    .\\D.: '
    shell_config.prompt_out = 'Out<\\#>: '
else:
    print "Running nested copies of IPython."
    print "The prompts for the nested copy have been modified"
    cfg = Config()
    nested = 1

# First import the embeddable shell class
from IPython.frontend.terminal.embed import InteractiveShellEmbed

# 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 = InteractiveShellEmbed(config=cfg,
                                banner1 = 'Dropping into IPython',
                                exit_msg = 'Leaving Interpreter, back to program.')

# Make a second instance, you can have as many as you want.
cfg2 = cfg.copy()
shell_config = cfg2.InteractiveShellEmbed
shell_config.prompt_in1 = 'In2<\\#>: '
if not nested:
    shell_config.prompt_in1 = 'In2<\\#>: '
    shell_config.prompt_in2 = '    .\\D.: '
```

```
    shell_config.prompt_out = 'Out<\\#>: '
ipshell2 = InteractiveShellEmbed(config=cfg,
                                banner1 = '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:

# InteractiveShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
# messages) is available as get_ipython().banner in case you want it.

# InteractiveShellEmbed 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
# by setting the banner and exit_msg attributes.

# 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.banner = 'Entering interpreter - New Banner'
ipshell.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(dummy_mode = True
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=False)

# Reactivate the IPython shell
ipshell(dummy_mode = False

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:
    get_ipython
except NameError:
    banner=exit_msg=''
else:
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'

# First import the embed function
```

```
from IPython.frontend.terminal.embed import InteractiveShellEmbed
# Now create the IPython shell instance. Put ipshell() anywhere in your code
# where you want it to open.
ipshell = InteractiveShellEmbed(banner1=banner, exit_msg=exit_msg)

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

from IPython import embed
# Now embed() 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:
    get_ipython
except NameError:
    from IPython.frontend.terminal.embed import InteractiveShellEmbed
    ipshell = InteractiveShellEmbed()
    # 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> *****
```

4.3.5 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.

4.3.6 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 Python or IPython prompts

IPython is smart enough to filter out input prompts, be they plain Python ones (>>> and . . .) or IPython ones (In [N]: and “...:”). You can therefore copy and paste from existing interactive sessions without worry.

The following is a ‘screenshot’ of how things work, copying an example from the standard Python tutorial:

```
In [1]: >>> # Fibonacci series:  
  
In [2]: ... # the sum of two elements defines the next  
  
In [3]: ... a, b = 0, 1
```

```
In [4]: >>> while b < 10:
...:     ...     print b
...:     ...     a, b = b, a+b
...:
1
1
2
3
5
8
```

And pasting from IPython sessions works equally well:

```
In [1]: In [5]: def f(x):
...:     ...:     "A simple function"
...:     ...:     return x**2
...:     ...:

In [2]: f(3)
Out[2]: 9
```

4.3.7 GUI event loop support

New in version 0.11: The `%gui` magic and `IPython.lib.inputhook`.

Warning: All GUI support with the `%gui` magic, described in this section, applies only to the plain terminal IPython, *not* to the Qt console. The Qt console currently only supports GUI interaction via the `--pylab` flag, as explained [in the matplotlib section](#).

We intend to correct this limitation as soon as possible, you can track our progress at issue #643_.

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 thread-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 simple use the `%gui` magic as follows:

```
%gui [GUINAME]
```

With no arguments, `%gui` removes all GUI support. Valid GUINAME arguments are `wx`, `qt4`, `gtk` and `tk`.

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

```
%gui 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` and `IPython.lib.guisupport` modules. 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. As indicated in the warning above, we plan on improving the integration of GUI event loops with the standalone kernel used by the Qt console and other frontends (issue [643](#)).

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 support for this. Since the details vary slightly between toolkits, we point you to the various examples in our source directory `docs/examples/lib` that demonstrate these capabilities.

Warning: The WX version of this is currently broken. While `--pylab=wx` works fine, standalone WX apps do not. See <https://github.com/ipython/ipython/issues/645> for details of our progress on this issue.

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.

PyQt and PySide

When you use `--gui=qt` or `--pylab=qt`, IPython can work with either PyQt4 or PySide. There are three options for configuration here, because PyQt4 has two APIs for `QString` and `QVariant` - v1, which is the default on Python 2, and the more natural v2, which is the only API supported by PySide. v2 is also the default for PyQt4 on Python 3. IPython’s code for the `QtConsole` uses v2, but you can still use any interface in your code, since the Qt frontend is in a different process.

The default will be to import PyQt4 without configuration of the APIs, thus matching what most applications would expect. It will fall back of PySide if PyQt4 is unavailable.

If specified, IPython will respect the environment variable `QT_API` used by ETS. ETS 4.0 also works with both PyQt4 and PySide, but it requires PyQt4 to use its v2 API. So if `QT_API=pyside` PySide will be used, and if `QT_API=pyqt` then PyQt4 will be used *with the v2 API* for `QString` and `QVariant`, so ETS codes like MayaVi will also work with IPython.

If you launch IPython in `pylab` mode with `ipython --pylab=qt`, then IPython will ask matplotlib which Qt library to use (only if `QT_API` is *not set*), via the ‘`backend.qt4`’ rcParam. If matplotlib is version 1.0.1 or older, then IPython will always use PyQt4 without setting the v2 APIs, since neither v2 PyQt nor PySide work.

Warning: Note that this means for ETS 4 to work with PyQt4, `QT_API` *must* be set to work with IPython's qt integration, because otherwise PyQt4 will be loaded in an incompatible mode. It also means that you must *not* have `QT_API` set if you want to use `--gui=qt` with code that requires PyQt4 API v1.

4.3.8 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.

To start IPython with matplotlib support, use the `--pylab` switch. If no arguments are given, IPython will automatically detect your choice of matplotlib backend. You can also request a specific backend with `--pylab=backend`, where `backend` must be one of: 'tk', 'qt', 'wx', 'gtk', 'osx'.

4.3.9 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.lib.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.lib 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, see `IPython.embed()` for details.

4.4 IPython as a system shell

Warning: As of the 0.11 version of IPython, most of the APIs used by the shell profile have been changed, so the profile currently does very little beyond changing the IPython prompt. To help restore the shell profile to past functionality described here, the old code is found in `IPython/deathrow`, which needs to be updated to use the APIs in 0.11.

4.4.1 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.

4.4.2 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.

4.4.3 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 automagic 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.

4.4.4 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=/sbin; - prepend /sbin; 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, bzr, 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'.

4.4.5 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.

4.4.6 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 = 12.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')
```

4.4.7 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’.

4.4.8 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.

4.5 A Qt Console for IPython

We now have a version of IPython, using the new two-process *ZeroMQ Kernel*, running in a `PyQt` GUI. This is a very lightweight widget that largely feels like a terminal, but provides a number of enhancements only possible in a GUI, such as inline figures, proper multiline editing with syntax highlighting, graphical calltips, and much more.

To get acquainted with the Qt console, type `%guiref` to see a quick introduction of its main features.

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

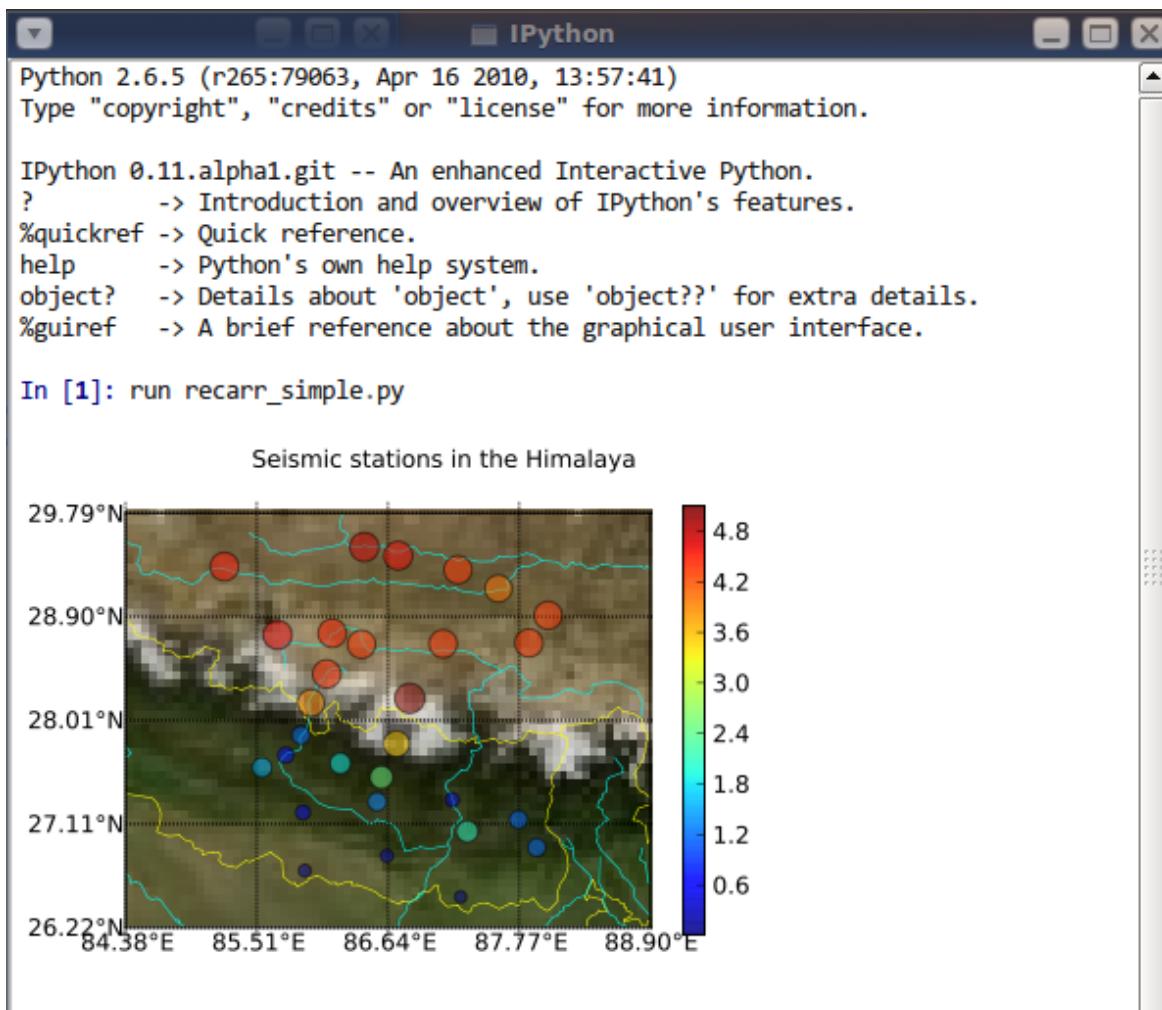
Tip: Since the Qt console tries hard to behave like a terminal, by default it immediately executes single lines of input that are complete. If you want to force multiline input, hit `:key:'Ctrl-Enter'` at the end of the first line instead of `:key:'Enter'`, and it will open a new line for input. At any point in a multiline block, you can force its execution (without having to go to the bottom) with `:key:'Shift-Enter'`.

4.5.1 %loadpy

The new `%loadpy` magic 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/cont
```

```
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()
```



The screenshot shows the IPython 0.11 interface with the title bar "IPython". The console window displays Python 2.6.5 information and a list of help commands. Below this, In [1] shows the command to run a script. The main area displays a map titled "Seismic stations in the Himalaya" with a color scale from 0.6 (blue) to 4.8 (red). The map shows station locations as colored circles and contour lines.

```

Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
Type "copyright", "credits" or "license" for more information.

IPython 0.11.alpha1.git -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
%uiref     -> A brief reference about the graphical user interface.

In [1]: run recarr_simple.py

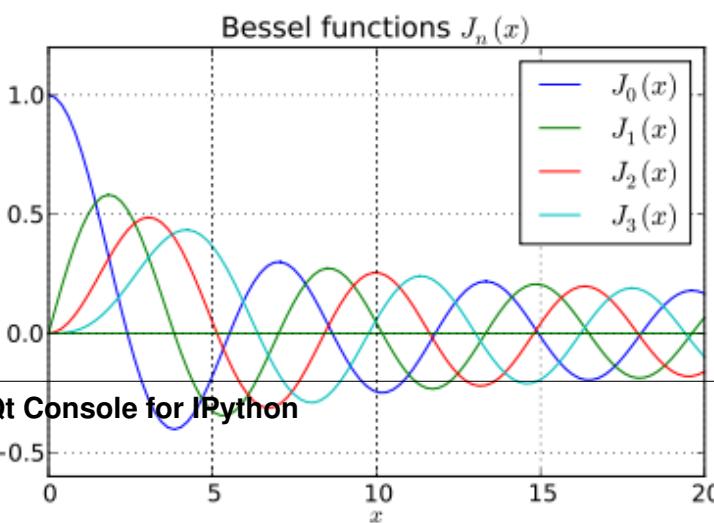
  Seismic stations in the Himalaya


```

```

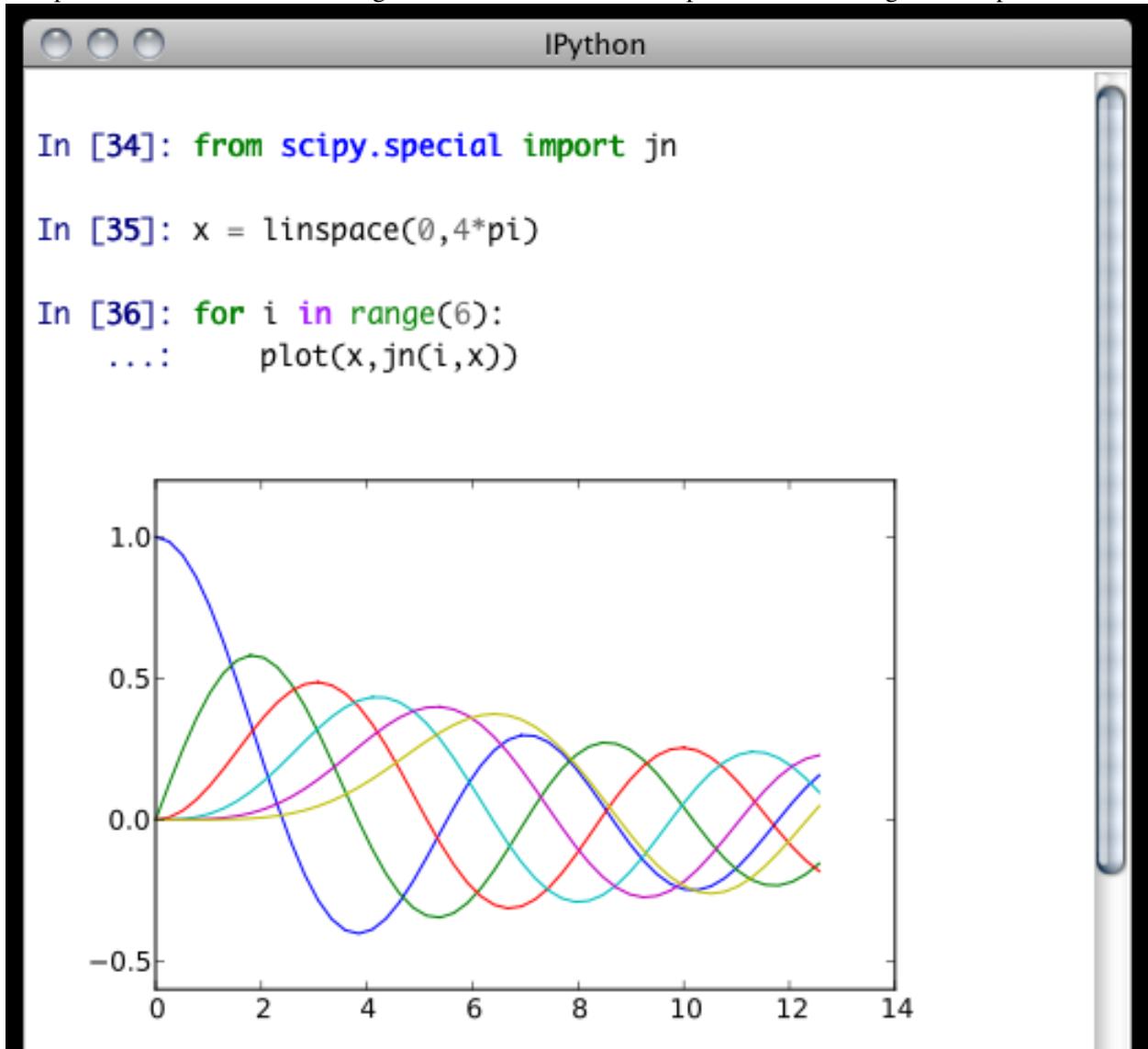
In [2]: from scipy import special as sp
....: x = linspace(0, 20, 100)
....: for n in range(4):
....:     y = sp.jn(n, x)
....:     plot(x, y, label=r'$J_{%s}(x)$' % n)
....: axhline(0, color='green', label='_nolegend_')
....: grid()
....: legend()
....: xlabel('$x$')
....: title(r'Bessel functions $J_n(x)$')
Out[2]: <matplotlib.text.Text object at 0x7fcddc1795d0>

```



4.5.2 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.



`display()`

An additional function, `display()`, will be added to the global namespace if you specify the `--pylab` option at the command line. The IPython display system provides a mechanism for specifying PNG or SVG (and more) representations of objects for GUI frontends. By default, IPython registers convenient PNG and SVG renderers for matplotlib figures, so you can embed them in your document by calling `display()` on one or more of them. This is especially useful for `saving` your work.

```
In [5]: plot(range(5)) # plots in the matplotlib window  
In [6]: display(gcf()) # embeds the current figure in the qtconsole  
In [7]: display(*getfigs()) # embeds all active figures in the qtconsole
```

If you have a reference to a matplotlib figure object, you can always display that specific figure:

```
In [1]: f = figure()  
  
In [2]: plot(rand(100))  
Out[2]: []  
  
In [3]: display(f)  
  
# Plot is shown here  
In [4]: title('A title')  
Out[4]: <matplotlib.text.Text at 0x7fc6ac023450>  
  
In [5]: display(f)
```

--pylab=inline

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

4.5.3 Saving and Printing

IPythonQt has the ability to save your current session, as either HTML or XHTML. If you have been using `display()` 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, which are used by default, but if you pass the `--plain` flag, saving will not be available to you.

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.

4.5.4 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 used in the terminal shell.

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=linux`, which uses the ‘monokai’ theme by default:

The screenshot shows the IPython Qt console window titled "IPython". It displays three code cells:

- 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; }
```

4.5.5 Fonts

The QtConsole has configurable via the ConsoleWidget. To change these, set the `font_family` or `font_size` traits of the ConsoleWidget. For instance, to use 9pt Anonymous Pro:

```
$> ipython qtconsole --ConsoleWidget.font_family="Anonymous Pro" --ConsoleWidget.font_size=9
```

4.5.6 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:

```
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing --shell=60690 --iopub=44045 --stdin=38323 --hb=41797
```

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.

4.5.7 Qt and the QtConsole

An important part of working with the QtConsole when you are writing your own Qt code is to remember that user code (in the kernel) is *not* in the same process as the frontend. This means that there is not necessarily any Qt code running in the kernel, and under most normal circumstances there isn't. If, however, you specify `--pylab=qt` at the command-line, then there *will* be a `QCoreApplication` instance running in the kernel process along with user-code. To get a reference to this application, do:

```
from PyQt4 import QtCore
app = QtCore.QCoreApplication.instance()
# app will be None if there is no such instance
```

A common problem listed in the PyQt4 [Gotchas](#) is the fact that Python's garbage collection will destroy Qt objects (Windows, etc.) once there is no longer a Python reference to them, so you have to hold on to them. For instance, in:

```
def make_window():
    win = QtGui.QMainWindow()

def make_and_return_window():
    win = QtGui.QMainWindow()
    return win
```

`make_window()` will never draw a window, because garbage collection will destroy it before it is drawn, whereas `make_and_return_window()` lets the caller decide when the window object should be destroyed. If, as a developer, you know that you always want your objects to last as long as the process, you can attach them to the `QApplication` instance itself:

```
# do this just once:
app = QtCore.QCoreApplication.instance()
app.references = set()
# then when you create Windows, add them to the set
def make_window():
    win = QtGui.QMainWindow()
    app.references.add(win)
```

Now the QApplication itself holds a reference to `win`, so it will never be garbage collected until the application itself is destroyed.

4.5.8 Regressions

There are some features, where the qt console lags behind the Terminal frontend:

- **!cmd input: Due to our use of pexpect, we cannot pass input to subprocesses** launched using the ‘!’ escape, so you should never call a command that requires interactive input. For such cases, use the terminal IPython. This will not be fixed, as abandoning pexpect would significantly degrade the console experience.
- Use of \b and \r characters in the console: these are control characters that allow the cursor to move backwards on a line, and are used to display things like in-place progress bars in a terminal. We currently do not support this, but it is being tracked as issue [629](#).

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 `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.

Tip: At the SciPy 2011 conference in Austin, Min Ragan-Kelley presented a complete 4-hour tutorial on the use of these features, and all the materials for the tutorial are now [available online](#). That tutorial provides an excellent, hands-on oriented complement to the reference documentation presented here.

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/profile_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/profile_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(sshserver='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.

If you are running engines on multiple machines, you will likely need to instruct the controller to listen for connections on an external interface. This can be done by specifying the `ip` argument on the command-line, or the `HubFactory.ip` configurable in `ipcontroller_config.py`.

If your machines are on a trusted network, you can safely instruct the controller to listen on all public interfaces with:

```
$> ipcontroller --ip=*
```

Or you can set the same behavior as the default by adding the following line to your `ipcontroller_config.py`:

```
c.HubFactory.ip = '*'

---


```

Note: Due to the lack of security in ZeroMQ, the controller will only listen for connections on localhost by default. If you see Timeout errors on engines or clients, then the first thing you should check is the ip address the controller is listening on, and make sure that it is visible from the timing out machine.

See Also:

Our notes on security in the new parallel computing code.

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`. The controller must be instructed to listen on an interface visible to the engine machines, via the `ip` command-line argument or `HubFactory.ip` in `ipcontroller_config.py`.
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/profile_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/profile_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 **mpiexec** 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/profile_<name>/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, do:

```
$ ipcluster -h
```

5.2.3 Configuring an IPython cluster

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

```
$ ipython profile create --parallel --profile=myprofile
```

This will create the directory `IPYTHONDIR/profile_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 `profile=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/profile_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 **ssh**, **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:

```
$ ipython profile create --parallel --profile=mpi
```

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

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

```
c.IPClusterEngines.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 --profile=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.IPClusterStart.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 or call `MPI_Init()`. However, older MPI implementations actually require each process to call `MPI_Init()` upon starting. The easiest way of having this done is to install the `mpi4py` [mpi4py] package and then 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:

```
$ ipython profile create --parallel --profile=pbs
```

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

```
c.IPClusterStart.controller_launcher = \
    'IPython.parallel.apps.launcher.PBSControllerLauncher'
c.IPClusterEngines.engine_launcher = \
    'IPython.parallel.apps.launcher.PBSEngineSetLauncher'
```

Note: Note that the configurable is IPClusterEngines for the engine launcher, and IPClusterStart for the controller launcher. This is because the start command is a subclass of the engine command, adding a controller launcher. Since it is a subclass, any configuration made in IPClusterEngines is inherited by IPClusterStart unless it is overridden.

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/mpexec -n {n} ipengine --profile-dir={profile_dir}
```

There are a few important points about this template:

1. This template will be rendered at runtime using IPython's `EvalFormatter`. This is simply a subclass of `string.Formatter` that allows simple expressions on keys.
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 `{n}` and `{profile_dir}` variables passed to the formatter. 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. Any options to `ipengine` can be given in the batch script template, or in `ipengine_config.py`.
4. 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 --profile-dir={profile_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.IPClusterEngines.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.HubFactory.ip = '*'  
  
You can now run the cluster with:
```

```
$ ipcluster start --profile=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 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:

```
$ ipython profile create --parallel --profile=ssh
```

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

```
c.IPClusterEngines.engine_launcher = \  
    'IPython.parallel.apps.launcher.SSHEngineSetLauncher'  
    # and if the Controller is also to be remote:  
c.IPClusterStart.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 'profile-dir' argument.
# c.SSHControllerLauncher.program_args = ['--reuse', '--ip=*', '--profile-dir=/path/to/cd']
```

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, ['--profile-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 = ['--profile-dir=/path/to/profile_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 - This is a regression from 0.10, which moved connection files around with `scp`. This will be improved, but not before 0.11 release.

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/profile_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 `--reuse`), 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**. The controller must be instructed to listen on an interface visible to the engine machines, via the `ip` command-line argument or `HubFactory.ip` in `ipcontroller_config.py`.
2. Copy `ipcontroller-engine.json` from `~/.ipython/profile_<name>/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/profile_<name>/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/profile_<name>/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 *--reuse* flag, so that the connection information in the JSON files remains accurate:

```
$ ipcontroller --reuse
```

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/profile_<name>/log`. Sending the log files to us will often help us to debug any problems.

Configuring *ipcontroller*

The IPython Controller takes its configuration from the file `ipcontroller_config.py` in the active profile directory.

Ports and addresses

In many cases, you will want to configure the Controller's network identity. By default, the Controller listens only on loopback, which is the most secure but often impractical. To instruct the controller to listen on a specific interface, you can set the `HubFactory.ip` trait. To listen on all interfaces, simply specify:

```
c.HubFactory.ip = '*'
```

When connecting to a Controller that is listening on loopback or behind a firewall, it may be necessary to specify an SSH server to use for tunnels, and the external IP of the Controller. If you specified that the HubFactory listen on loopback, or all interfaces, then IPython will try to guess the external IP. If you are on a system with VM network devices, or many interfaces, this guess may be incorrect. In these cases, you will want to specify the 'location' of the Controller. This is the IP of the machine the Controller is on, as seen by the clients, engines, or the SSH server used to tunnel connections.

For example, to set up a cluster with a Controller on a work node, using ssh tunnels through the login node, an example `ipcontroller_config.py` might contain:

```
# allow connections on all interfaces from engines
# engines on the same node will use loopback, while engines
# from other nodes will use an external IP
c.HubFactory.ip = '*'

# you typically only need to specify the location when there are extra
# interfaces that may not be visible to peer nodes (e.g. VM interfaces)
c.HubFactory.location = '10.0.1.5'
# or to get an automatic value, try this:
import socket
ex_ip = socket.gethostbyname_ex(socket.gethostname())[-1][0]
c.HubFactory.location = ex_ip

# now instruct clients to use the login node for SSH tunnels:
c.HubFactory.ssh_server = 'login.mycluster.net'
```

After doing this, your `ipcontroller-client.json` file will look something like this:

```
{
    "url": "tcp://*:43447",
    "exec_key": "9c7779e4-d08a-4c3b-ba8e-db1f80b562c1",
    "ssh": "login.mycluster.net",
    "location": "10.0.1.5"
}
```

Then this file will be all you need for a client to connect to the controller, tunneling SSH connections through `login.mycluster.net`.

Database Backend

The Hub stores all messages and results passed between Clients and Engines. For large and/or long-running clusters, it would be unreasonable to keep all of this information in memory. For this reason, we have two database backends: [\[MongoDB\]](#) via [PyMongo](#), and SQLite with the stdlib `sqlite`.

MongoDB is our design target, and the dict-like model it uses has driven our design. As far as we are concerned, BSON can be considered essentially the same as JSON, adding support for binary data and datetime objects, and any new database backend must support the same data types.

See Also:

[MongoDB BSON doc](#)

To use one of these backends, you must set the `HubFactory.db_class` trait:

```
# for a simple dict-based in-memory implementation, use dictdb
# This is the default and the fastest, since it doesn't involve the filesystem
c.HubFactory.db_class = 'IPython.parallel.controller.dictdb.DictDB'

# To use MongoDB:
c.HubFactory.db_class = 'IPython.parallel.controller.mongodb.MongoDB'
```

```
# and SQLite:  
c.HubFactory.db_class = 'IPython.parallel.controller.sqlitedb.SQLiteDB'
```

When using the proper databases, you can actually allow for tasks to persist from one session to the next by specifying the MongoDB database or SQLite table in which tasks are to be stored. The default is to use a table named for the Hub's Session, which is a UUID, and thus different every time.

```
# To keep persistant task history in MongoDB:  
c.MongoDB.database = 'tasks'
```

```
# and in SQLite:  
c.SQLiteDB.table = 'tasks'
```

Since MongoDB servers can be running remotely or configured to listen on a particular port, you can specify any arguments you may need to the PyMongo Connection:

```
# positional args to pymongo.Connection  
c.MongoDB.connection_args = []  
  
# keyword args to pymongo.Connection  
c.MongoDB.connection_kwargs = {}
```

Configuring *ipengine*

The IPython Engine takes its configuration from the file ipengine_config.py

The Engine itself also has some amount of configuration. Most of this has to do with initializing MPI or connecting to the controller.

To instruct the Engine to initialize with an MPI environment set up by mpi4py, add:

```
c.MPI.use = 'mpi4py'
```

In this case, the Engine will use our default mpi4py init script to set up the MPI environment prior to execution. We have default init scripts for mpi4py and pytrilinos. If you want to specify your own code to be run at the beginning, specify *c.MPI.init_script*.

You can also specify a file or python command to be run at startup of the Engine:

```
c.IPEngineApp.startup_script = u'/path/to/my/startup.py'  
  
c.IPEngineApp.startup_command = 'import numpy, scipy, mpi4py'
```

These commands/files will be run again, after each

It's also useful on systems with shared filesystems to run the engines in some scratch directory. This can be set with:

```
c.IPEngineApp.work_dir = u'/path/to/scratch/'
```

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/profile_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/asyncresult.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]
```

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/user/<ipython-input-10-5d56b303a66c> in <module>()
----> 1 dview.execute('1/0')

/path/to/site-packages/IPython/parallel/client/view.pyc in execute(self, code, targets, block)
    591             default: self.block
    592             """
--> 593             return self._really_apply(util._execute, args=(code,), block=block, targets=targets)
    594
    595     def run(self, filename, targets=None, block=None):

/home/user/<string> in _really_apply(self, f, args, kwargs, targets, block, track)

/path/to/site-packages/IPython/parallel/client/view.pyc in sync_results(f, self, *args, **kwargs)
    55     def sync_results(f, self, *args, **kwargs):
    56         """sync relevant results from self.client to our results attribute."""
--> 57         ret = f(self, *args, **kwargs)
    58         delta = self.outstanding.difference(self.client.outstanding)
    59         completed = self.outstanding.intersection(delta)

/home/user/<string> in _really_apply(self, f, args, kwargs, targets, block, track)
```

```
/path/to/site-packages/IPython/parallel/client/view.pyc in save_ids(f, self, *args, **kwargs)
    44     n_previous = len(self.client.history)
    45     try:
--> 46         ret = f(self, *args, **kwargs)
    47     finally:
    48         nmsgs = len(self.client.history) - n_previous

/path/to/site-packages/IPython/parallel/client/view.pyc in _really_apply(self, f, args, kwargs)
    529         if block:
    530             try:
--> 531                 return ar.get()
    532             except KeyboardInterrupt:
    533                 pass

/path/to/site-packages/IPython/parallel/client/asyncresult.pyc in get(self, timeout)
    101             return self._result
    102         else:
--> 103             raise self._exception
    104         else:
    105             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 parallel.error.CompositeError, e:
....:     e.raise_exception()
....:
....:

-----
RemoteError                                                 Traceback (most recent call last)
/home/user/<ipython-input-17-8597e7e39858> in <module>()
      2     dview.execute('1/0')
      3 except CompositeError as e:
--> 4     e.raise_exception()

/path/to/site-packages/IPython/parallel/error.pyc in raise_exception(self, excid)
    266         raise IndexError("an exception with index %i does not exist"%excid)
    267     else:
--> 268         raise RemoteError(en, ev, etb, ei)
    269
    270

RemoteError: ZeroDivisionError(integer division or modulo by zero)
Traceback (most recent call last):
```

```
File "/path/to/site-packages/IPython/parallel/engine/streamkernel.py", line 330, in apply
    exec code in working,working
File "<string>", line 1, in <module>
File "/path/to/site-packages/IPython/parallel/util.py", line 354, in _execute
    exec code in globals()
File "<string>", line 1, in <module>
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/user/<ipython-input-10-5d56b303a66c> in <module>()
----> 1 dview.execute('1/0')

/path/to/site-packages/IPython/parallel/client/view.pyc in execute(self, code, targets, block)
  591         default: self.block
  592     """
--> 593     return self._really_apply(util._execute, args=(code,), block=block, targets=targets)
  594
  595     def run(self, filename, targets=None, block=None):

/home/user/<string> in _really_apply(self, f, args, kwargs, targets, block, track)

/path/to/site-packages/IPython/parallel/client/view.pyc in sync_results(f, self, *args, **kwargs)
  55 def sync_results(f, self, *args, **kwargs):
  56     """sync relevant results from self.client to our results attribute."""
--> 57     ret = f(self, *args, **kwargs)
  58     delta = self.outstanding.difference(self.client.outstanding)
  59     completed = self.outstanding.intersection(delta)

/home/user/<string> in _really_apply(self, f, args, kwargs, targets, block, track)

/path/to/site-packages/IPython/parallel/client/view.pyc in save_ids(f, self, *args, **kwargs)
  44     n_previous = len(self.client.history)
  45     try:
--> 46         ret = f(self, *args, **kwargs)
  47     finally:
  48         nmsgs = len(self.client.history) - n_previous

/path/to/site-packages/IPython/parallel/client/view.pyc in _really_apply(self, f, args, kwargs)
  529         if block:
  530             try:
--> 531                 return ar.get()
  532             except KeyboardInterrupt:
  533                 pass

/path/to/site-packages/IPython/parallel/client/asyncrequest.pyc in get(self, timeout)
  101         return self._result
  102     else:
--> 103         raise self._exception
```

```
104         else:
105             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/client/asyncresult.py(103) get()
    102         else:
--> 103             raise self._exception
    104         else:

ipdb> self._exception.<tab>
e.__class__          e.__getitem__        e.__new__           e.__setstate__
e.__delattr__        e.__getslice__       e.__reduce__        e.__str__
e.__dict__           e.__hash__          e.__reduce_ex__   e.__weakref__
e.__doc__            e.__init__          e.__repr__         e._get_engine_str
e.__getattribute__   e.__module__        e.__setattr__      e._get_traceback
ipdb> self._exception.print_tracebacks()
[0:apply]:
Traceback (most recent call last):
  File "/path/to/site-packages/IPython/parallel/engine/streamkernel.py", line 330, in apply
    exec code in working,working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/util.py", line 354, 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/engine/streamkernel.py", line 330, in apply
    exec code in working,working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/util.py", line 354, 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/engine/streamkernel.py", line 330, in apply
    exec code in working,working
  File "<string>", line 1, in <module>
  File "/path/to/site-packages/IPython/parallel/util.py", line 354, 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/engine/streamkernel.py", line 330, in apply  
    exec code in working,working  
  File "<string>", line 1, in <module>  
  File "/path/to/site-packages/IPython/parallel/util.py", line 354, in _execute  
    exec code in globals()  
  File "<string>", line 1, in <module>  
ZeroDivisionError: integer division or modulo by zero
```

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)  
/home/user/<ipython-input-21-8531eb3d26fb> in <module>()  
----> 1 ar.get()
```

```
/path/to/site-packages/IPython/parallel/client/asyncrequest.pyc in get(self, timeout)  
 101         return self._result  
 102     else:  
--> 103         raise self._exception  
 104     else:  
 105         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 `pymq` 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:

anyall 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 orAsyncResult 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 Retries and Resubmit

Retries

Another flag for tasks is *retries*. This is an integer, specifying how many times a task should be resubmitted after failure. This is useful for tasks that should still run if their engine was shutdown, or may have some statistical chance of failing. The default is to not retry tasks.

Resubmit

Sometimes you may want to re-run a task. This could be because it failed for some reason, and you have fixed the error, or because you want to restore the cluster to an interrupted state. For this, the Client has a `rc.resubmit()` method. This simply takes one or more `msg_ids`, and returns an `AsyncHubResult` for the result(s). You cannot resubmit a task that is pending - only those that have finished, either successful or unsuccessful.

5.4.6 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 `TaskScheduler.schemename` 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

5.4.7 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

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* Launchers in `ipcluster`, which will first start a controller and then a set of engines using `mpiexec`:

```
$ ipcluster start --n=4 --elauncher=MPIExecEngineSetLauncher
```

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, lets define a simply 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],  
    [recvBuf, MPI.DOUBLE],  
    op=MPI.SUM)  
return recvBuf
```

Now, start an IPython cluster:

```
$ ipcluster start --profile=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 IPython's Task Database

The IPython Hub stores all task requests and results in a database. Currently supported backends are: MongoDB, SQLite (the default), and an in-memory DictDB. The most common use case for this is clients requesting results for tasks they did not submit, via:

```
In [1]: rc.get_result(task_id)
```

However, since we have this DB backend, we provide a direct query method in the `client` for users who want deeper introspection into their task history. The `db_query()` method of the Client is modeled after MongoDB queries, so if you have used MongoDB it should look familiar. In fact, when the MongoDB

backend is in use, the query is relayed directly. However, when using other backends, the interface is emulated and only a subset of queries is possible.

See Also:

MongoDB query docs: <http://www.mongodb.org/display/DOCS/Querying>

`Client.db_query()` takes a dictionary query object, with keys from the TaskRecord key list, and values of either exact values to test, or MongoDB queries, which are dicts of the form: `{'operator' : 'argument(s)'}.` There is also an optional `keys` argument, that specifies which subset of keys should be retrieved. The default is to retrieve all keys excluding the request and result buffers. `db_query()` returns a list of TaskRecord dicts. Also like MongoDB, the `msg_id` key will always be included, whether requested or not.

TaskRecord keys:

| Key | Type | Description |
|-----------------------------|-----------------------------|---|
| <code>msg_id</code> | <code>uuid(bytes)</code> | The msg ID |
| <code>header</code> | <code>dict</code> | The request header |
| <code>content</code> | <code>dict</code> | The request content (likely empty) |
| <code>buffers</code> | <code>list(bytes)</code> | buffers containing serialized request objects |
| <code>submitted</code> | <code>datetime</code> | timestamp for time of submission (set by client) |
| <code>client_uuid</code> | <code>uuid(bytes)</code> | IDENT of client's socket |
| <code>engine_uuid</code> | <code>uuid(bytes)</code> | IDENT of engine's socket |
| <code>started</code> | <code>datetime</code> | time task began execution on engine |
| <code>completed</code> | <code>datetime</code> | time task finished execution (success or failure) on engine |
| <code>resubmitted</code> | <code>datetime</code> | time of resubmission (if applicable) |
| <code>result_header</code> | <code>dict</code> | header for result |
| <code>result_content</code> | <code>dict</code> | content for result |
| <code>result_buffers</code> | <code>list(bytes)</code> | buffers containing serialized request objects |
| <code>queue</code> | <code>bytes</code> | The name of the queue for the task ('mux' or 'task') |
| <code>pyin</code> | <code><unused></code> | Python input (unused) |
| <code>pyout</code> | <code><unused></code> | Python output (unused) |
| <code>pyerr</code> | <code><unused></code> | Python traceback (unused) |
| <code>stdout</code> | <code>str</code> | Stream of stdout data |
| <code>stderr</code> | <code>str</code> | Stream of stderr data |

MongoDB operators we emulate on all backends:

| Operator | Python equivalent |
|----------------------|---------------------|
| <code>'\$in'</code> | <code>in</code> |
| <code>'\$nin'</code> | <code>not in</code> |
| <code>'\$eq'</code> | <code>==</code> |
| <code>'\$ne'</code> | <code>!=</code> |
| <code>'\$ge'</code> | <code>></code> |
| <code>'\$gte'</code> | <code>>=</code> |
| <code>'\$le'</code> | <code><</code> |
| <code>'\$lte'</code> | <code><=</code> |

The DB Query is useful for two primary cases:

1. deep polling of task status or metadata
2. selecting a subset of tasks, on which to perform a later operation (e.g. wait on result, purge records, resubmit,...)

5.6.1 Example Queries

To get all msg_ids that are not completed, only retrieving their ID and start time:

```
In [1]: incomplete = rc.db_query({'complete' : None}, keys=['msg_id', 'started'])
```

All jobs started in the last hour by me:

```
In [1]: from datetime import datetime, timedelta
```

```
In [2]: hourago = datetime.now() - timedelta(1./24)
```

```
In [3]: recent = rc.db_query({'started' : {'$gte' : hourago },
                             'client_uuid' : rc.session.session})
```

All jobs started more than an hour ago, by clients *other than me*:

```
In [3]: recent = rc.db_query({'started' : {'$le' : hourago },
                             'client_uuid' : {'$ne' : rc.session.session}})
```

Result headers for all jobs on engine 3 or 4:

```
In [1]: uids = map(rc._engines.get, (3,4))
```

```
In [2]: hist34 = rc.db_query({'engine_uuid' : {'$in' : uids }, keys='result_header'})
```

5.7 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.

5.7.1 Process 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 *cluster*, and the hub and schedulers together are referred to as the *controller*.

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 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.

5.7.2 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).

5.7.3 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 with 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, [HMAC] digests are used to sign messages, using a shared key.

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 used to initialize an HMAC object, which digests all messages, and includes that digest as a signature and part of the message. Every message that is unpacked (on Controller, Engine, and Client) will also be digested by the receiver, ensuring that the sender's key is the same as the receiver's. No messages that do not contain this key are acted upon in any way. The key itself is never sent over the network.

There is exactly one shared 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/profile_<name>/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.

5.7.4 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} . For added security, users can have arbitrarily long keys.

Warning: If the attacker has gained enough access to intercept loopback connections on *either* the controller or client, then a duplicate message can be sent. To protect against this, recipients only allow each signature once, and consider duplicates invalid. However, the duplicate message could be sent to *another* recipient using the same key, and it would be considered valid.

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.

5.7.5 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.

5.7.6 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.

5.8 Getting started with Windows HPC Server 2008

Note: Not adapted to zmq yet

5.8.1 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 treads 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.org/documentation.html>).

5.8.2 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.

5.8.3 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.org>)

In addition, the following dependencies are needed to run the demos described in this document.

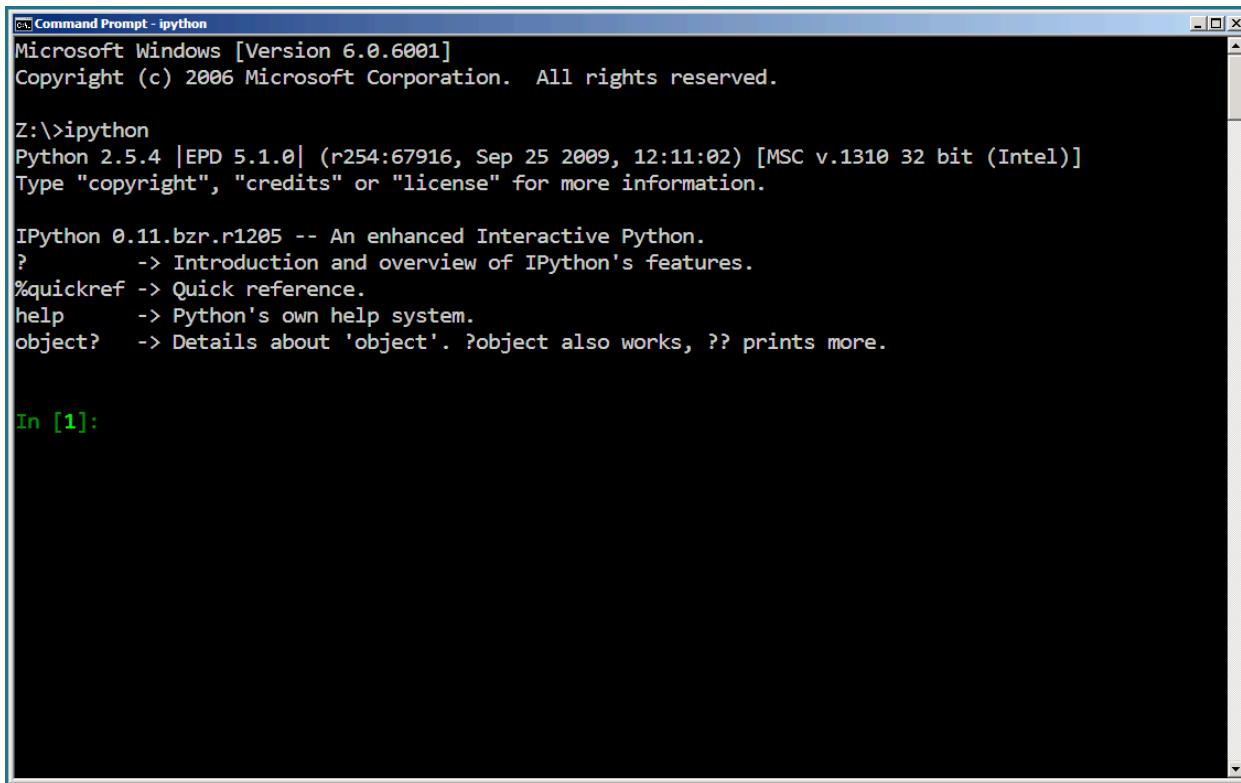
- NumPy and SciPy (<http://www.scipy.org>)
- Matplotlib (<http://matplotlib.sourceforge.net/>)

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.org>) and following the installation instructions.

Further details about installing IPython or its dependencies can be found in the online IPython documentation (<http://ipython.org/documentation.html>) 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:



The screenshot shows a Windows Command Prompt window titled "Command Prompt - ipython". The window displays the following text:

```

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]:

```

5.8.4 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.

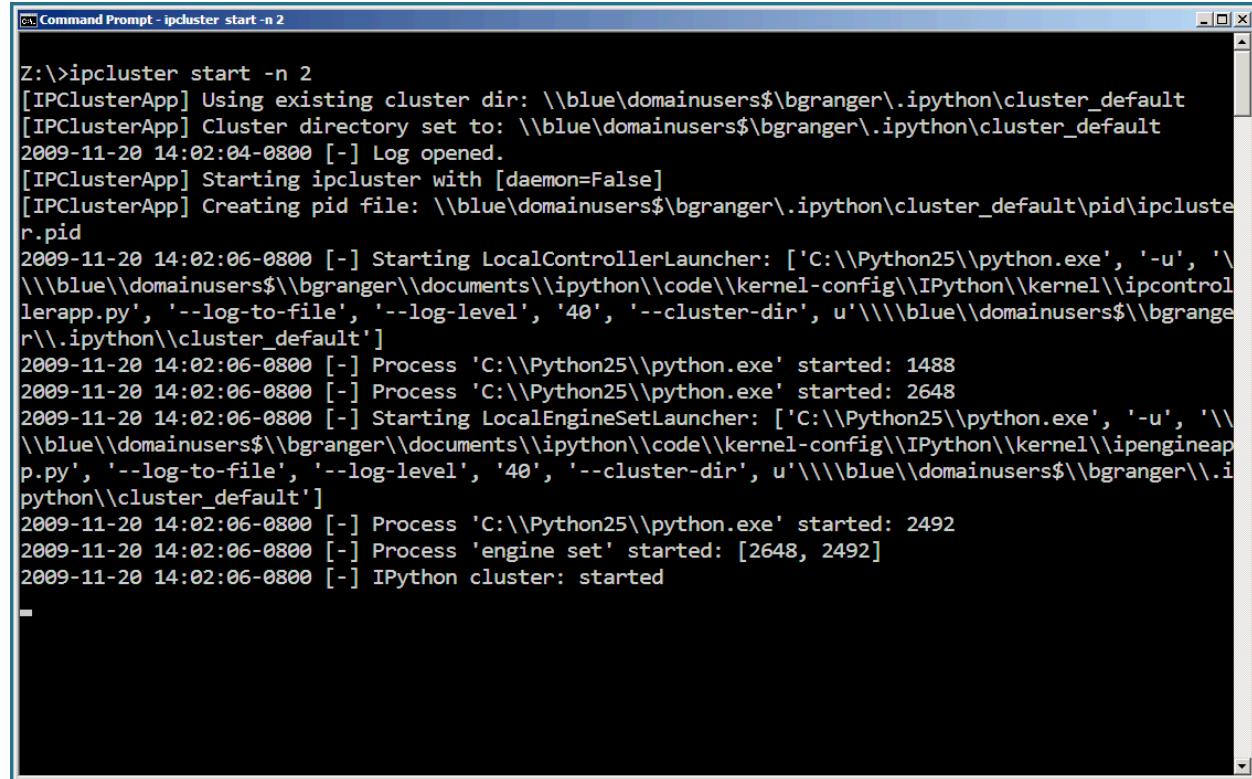
IPython has a command line program called **ipcluster** that automates all aspects of starting the controller and engines on the compute nodes. **ipcluster** has full support for the Windows HPC job scheduler, meaning that **ipcluster** can use this job scheduler to start the controller and engines.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



The screenshot shows a Windows Command Prompt window titled "Command Prompt - ipcluster start -n 2". The window contains the following text output:

```
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$\bgranger\.ipython\cluster_default\pid\ipcluste
r.pid
2009-11-20 14:02:06-0800 [-] Starting LocalControllerLauncher: ['C:\\Python25\\python.exe', '-u', '\\
\\blue\\domainusers$\\bgranger\\documents\\ipython\\code\\kernel-config\\IPython\\kernel\\ipcontrol
lerapp.py', '--log-to-file', '--log-level', '40', '--cluster-dir', u'\\\\blue\\domainusers$\\bgrange
r\\.ipython\\cluster_default']
2009-11-20 14:02:06-0800 [-] Process 'C:\\Python25\\python.exe' started: 1488
2009-11-20 14:02:06-0800 [-] Process 'C:\\Python25\\python.exe' started: 2648
2009-11-20 14:02:06-0800 [-] Starting LocalEngineSetLauncher: ['C:\\Python25\\python.exe', '-u', '\\
\\blue\\domainusers$\\bgranger\\documents\\ipython\\code\\kernel-config\\IPython\\kernel\\ipengin
eapp.py', '--log-to-file', '--log-level', '40', '--cluster-dir', u'\\\\blue\\domainusers$\\bgranger\\i
python\\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: `ipython profile create --parallel profile=mycluster`
2. Edit configuration files in the directory `.ipython\cluster_mycluster`
3. Start the cluster using: `ipcluser start profile=mycluster n=32`

Creating a cluster profile

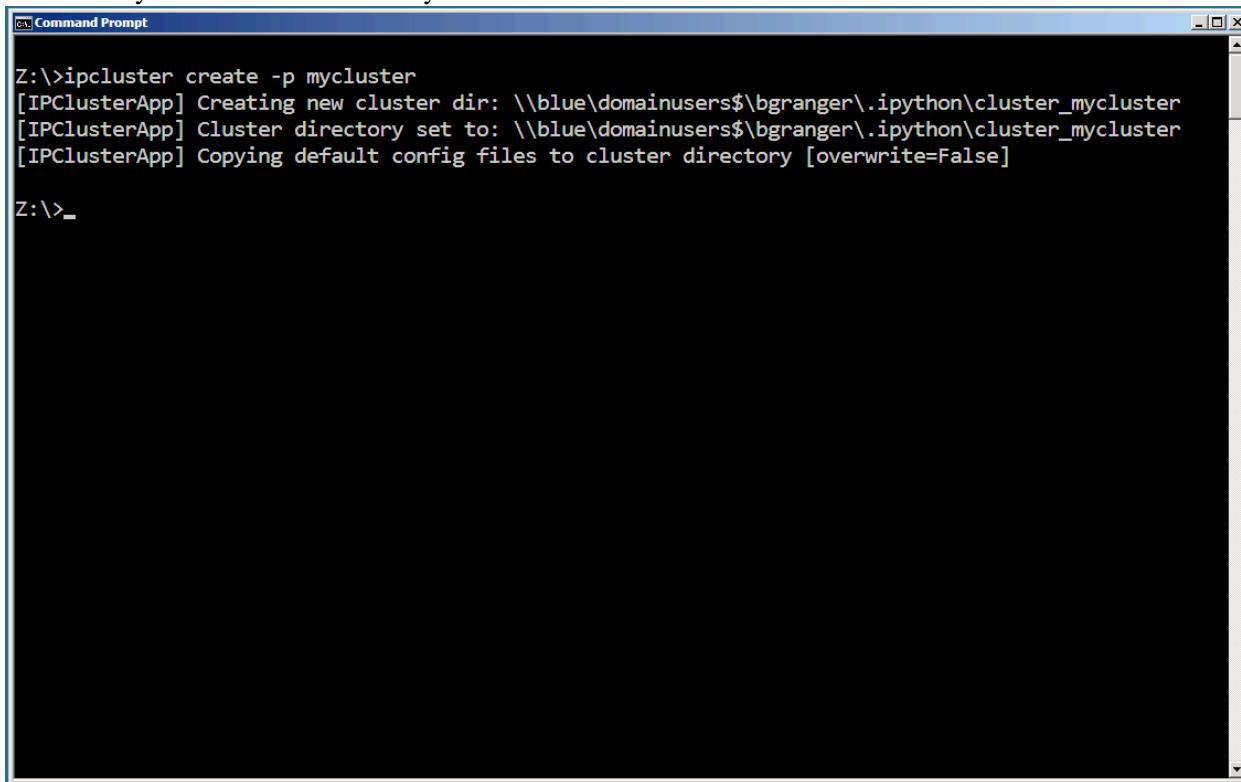
In most cases, you will have to create a cluster profile to use IPython on a cluster. A cluster profile is a name (like “mycluster”) that is associated with a particular cluster configuration. The profile name is used by **ipcluster** when working with the cluster.

Associated with each cluster profile is a cluster directory. This cluster directory is a specially named directory (typically located in the .ipython subdirectory of your home directory) that contains the configuration files for a particular cluster profile, as well as log files and security keys. The naming convention for cluster directories is: `profile_<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:

```
ipython profile create --parallel --profile=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.

A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text:

```
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:\>_
```

The window has a standard Windows title bar and scroll bars on the right side.

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.IPClusterStart.controller_launcher = \
    'IPython.parallel.apps.launcher.WindowsHPCControllerLauncher'
c.IPClusterEngines.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 --profile=mycluster --n=32
```

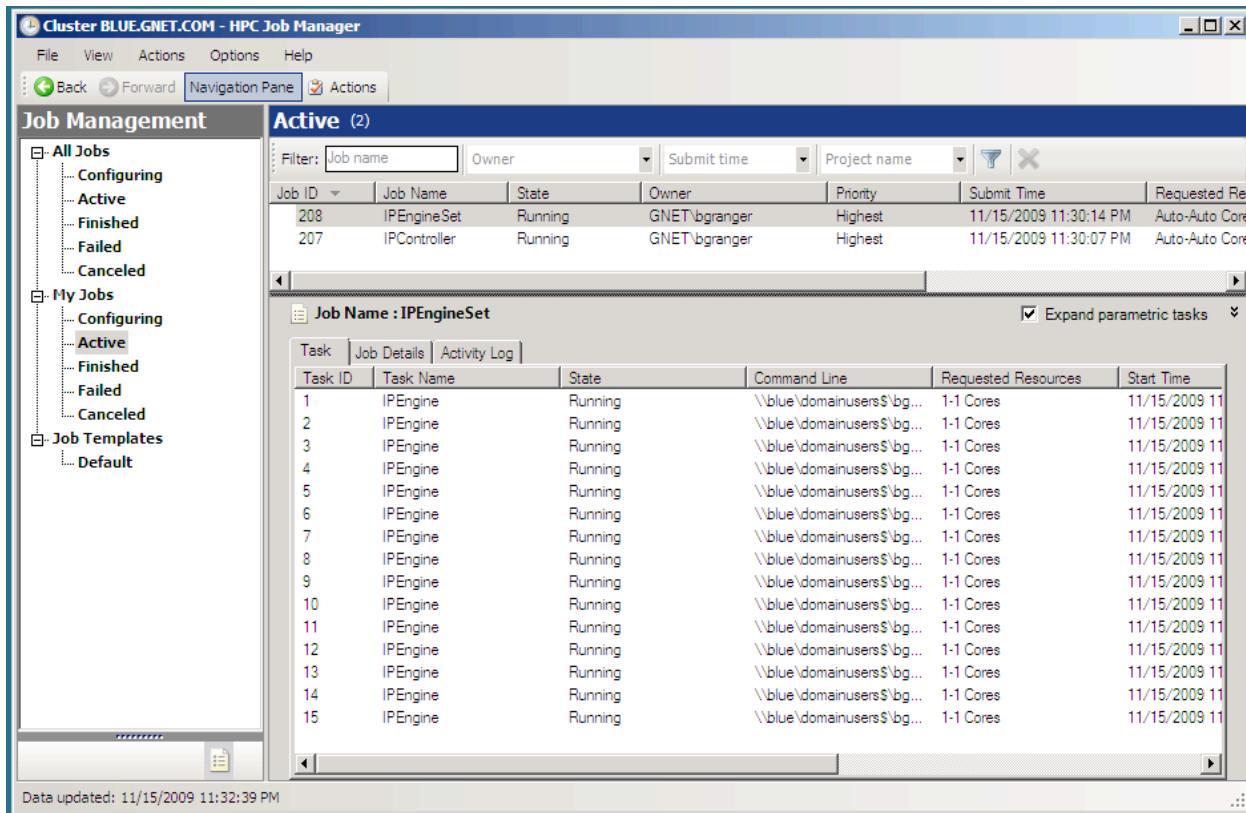
The `-n` option tells **ipcluster** how many engines to start (in this case 32). Stopping the cluster is as simple as typing Control-C.

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.



5.8.5 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:

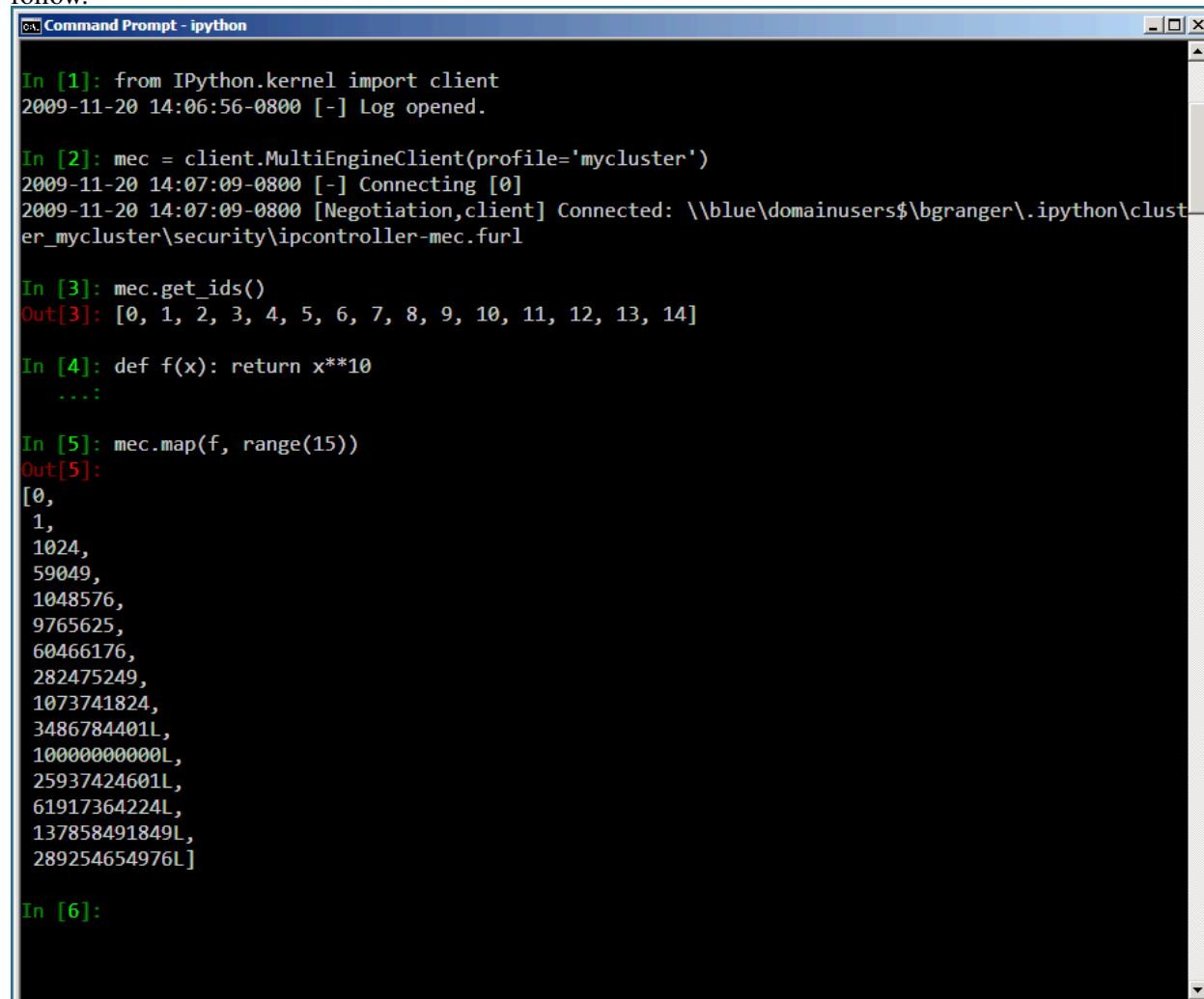
```
ipython
```

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, 67, 8, 9, 10, 11, 12, 13, 14]
In [4]: def f(x):
...:     return x**10
In [5]: mec.map(f, range(15)) # f is applied in parallel
Out[5]:
[0,
 1,
```

```
1024,
59049,
1048576,
9765625,
60466176,
282475249,
1073741824,
3486784401L,
100000000000L,
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.



The screenshot shows a Windows Command Prompt window titled "Command Prompt - ipython". The session history is as follows:

```
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,
 100000000000L,
 25937424601L,
 61917364224L,
 137858491849L,
 289254654976L]

In [6]:
```

5.9 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.

5.9.1 150 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
```

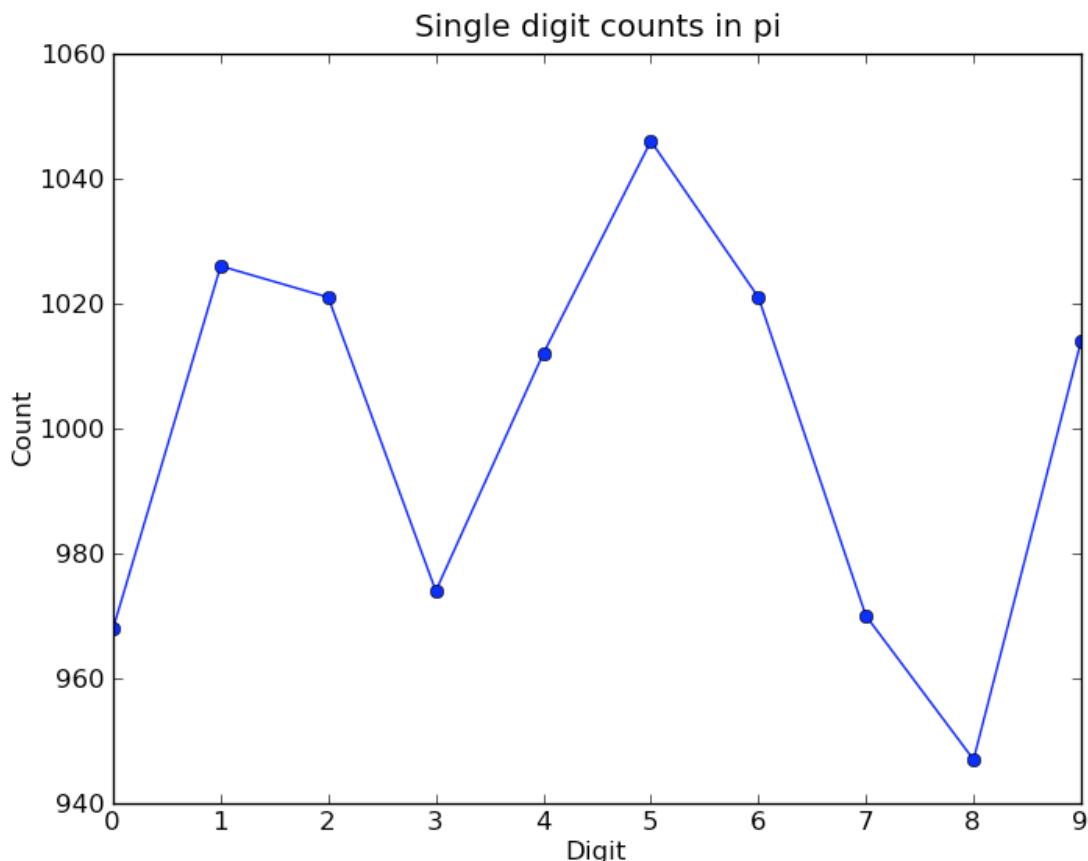
```
In [12]: run pidigits.py # load one_digit_freqs/plot_one_digit_freqs
```

```
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`:

```
def compute_two_digit_freqs(filename):
    """
    Read digits of pi from a file and compute the 2 digit frequencies.
    """
    d = txt_file_to_digits(filename)
    freqs = two_digit_freqs(d)
    return freqs

def reduce_freqs(freqlist):
    """
    Add up a list of freq counts to get the total counts.
    """
    allfreqs = np.zeros_like(freqlist[0])
    for f in freqlist:
        allfreqs += f
    return allfreqs
```

We will also use the `plot_two_digit_freqs()` function to plot the results. The code to run this calculation in parallel is contained in `docs/examples/newparallel/parallelpipi.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 looks like 16 (1 controller + 15 engines) in the OS. However, the maximum speedup we can observe is still only 8x.
2. With the file `parallelpipi.py` in your current working directory, open up IPython in pylab mode and type `run parallelpipi.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 `parallelpipi.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.%i02dof20'

# 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)

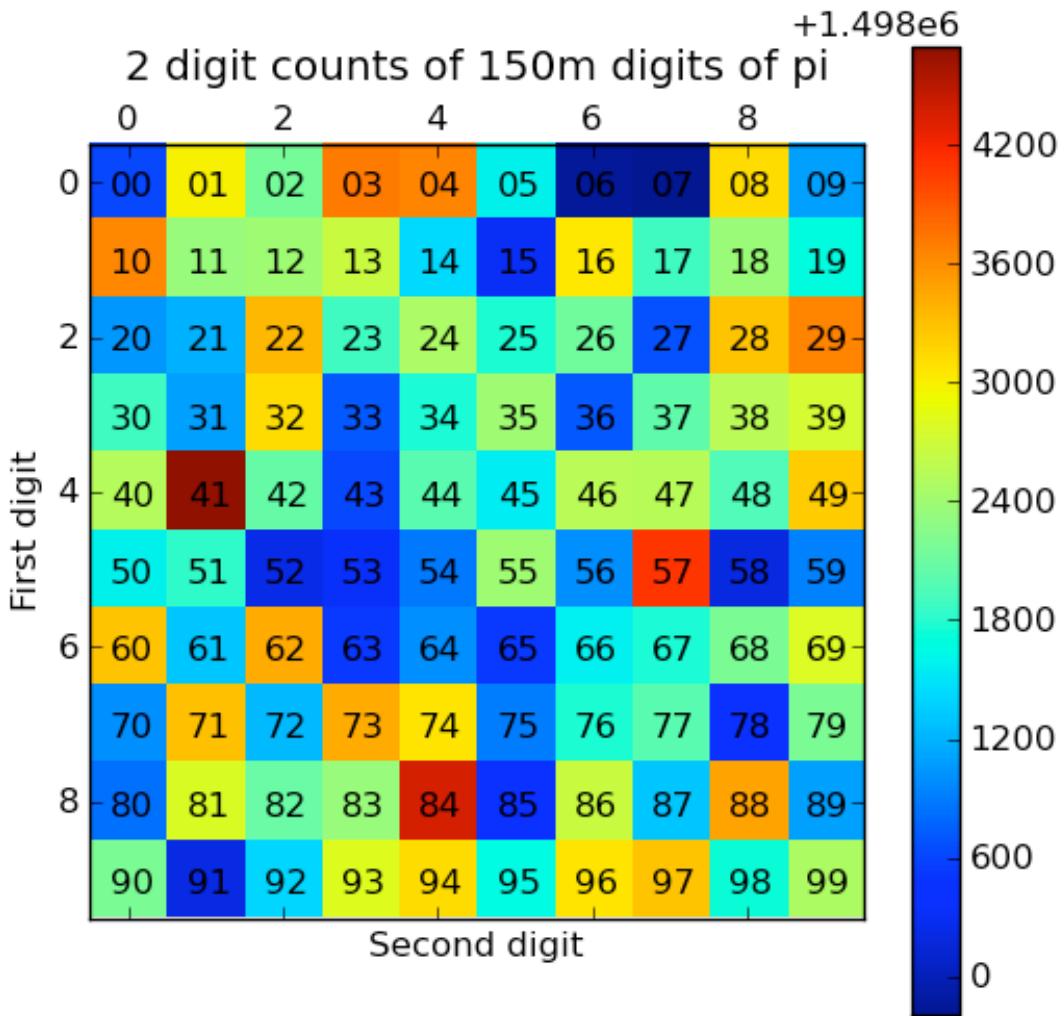
# This is the parallel calculation using the Client.map method
# which applies compute_two_digit_freqs to each file in files in parallel.
In [9]: freqs_all = v.map(compute_two_digit_freqs, files)

# 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.



5.9.2 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 `mpricer.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 `mpricer.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, ...]
```

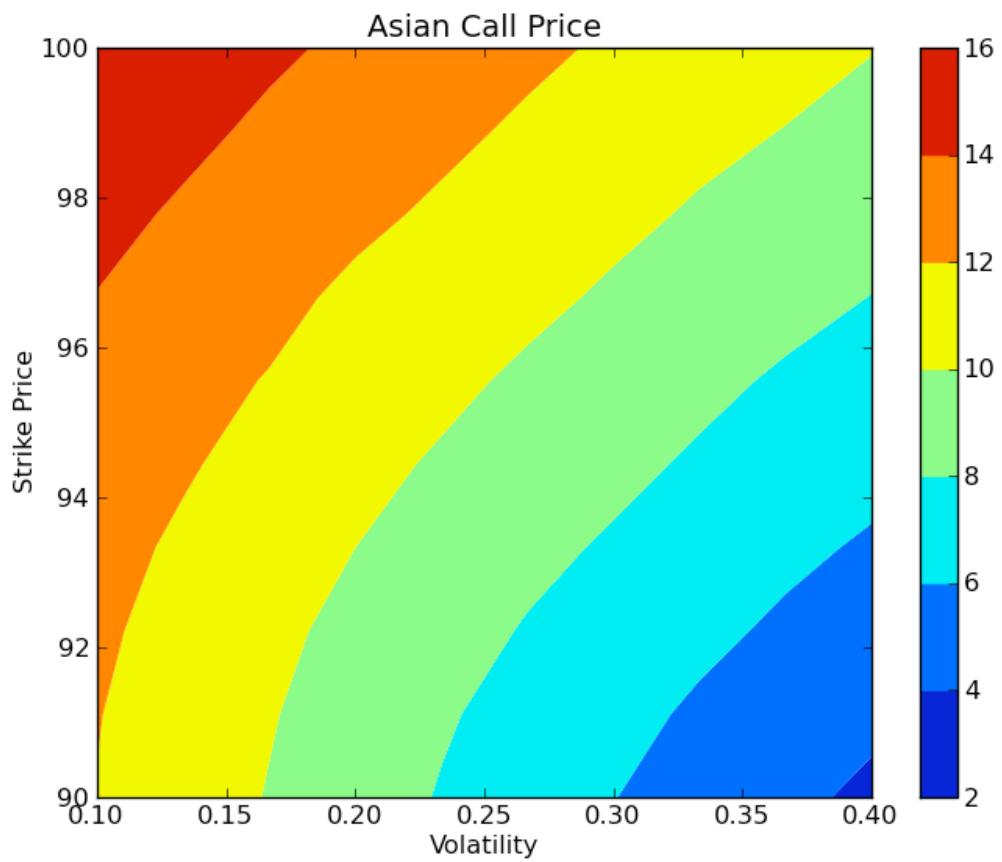
Once all the tasks have finished, the results can be plotted using the `plot_options()` function. Here we make contour plots of the Asian call and Asian put options as function of the volatility and strike price:

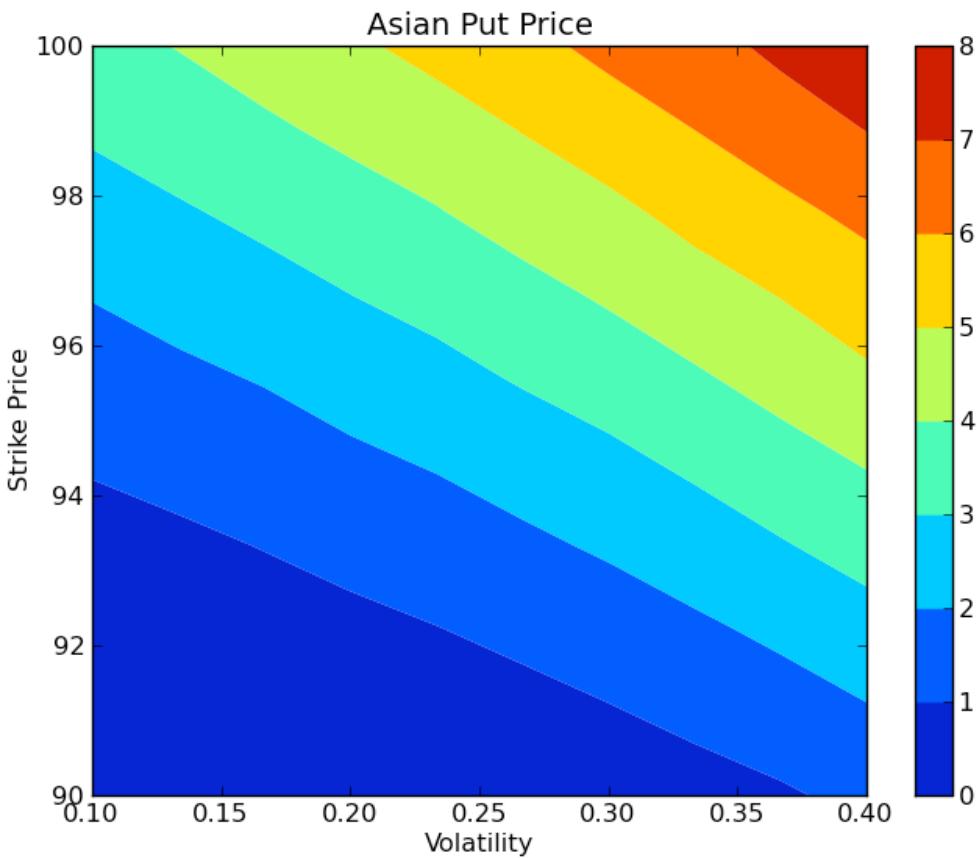
```
In [8]: plot_options(sigma_vals, K_vals, prices['acall'])
```

```
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.





5.9.3 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.

5.10 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`.

5.10.1 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.

5.10.2 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 `succes=True,failure=False`, the following cases would occur for each node’s failure:

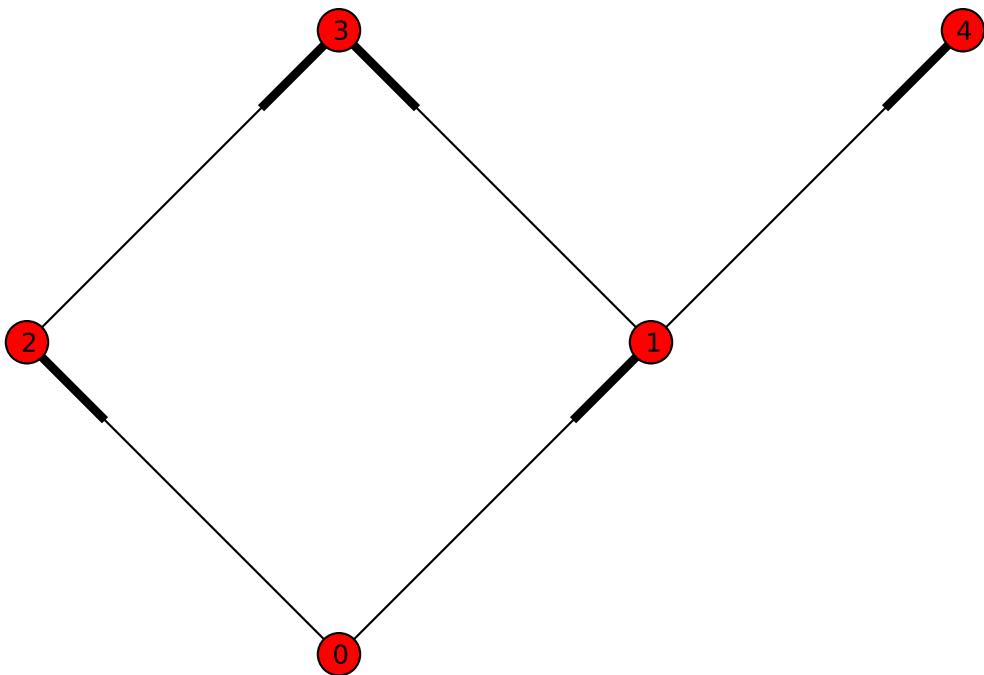
0. fails: all other tasks fail as Impossible
1. 2 can still succeed, but 3,4 are unreachable
2. 3 becomes unreachable, but 4 is unaffected
3. and 4. are terminal, and can have no effect on other nodes

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)
```

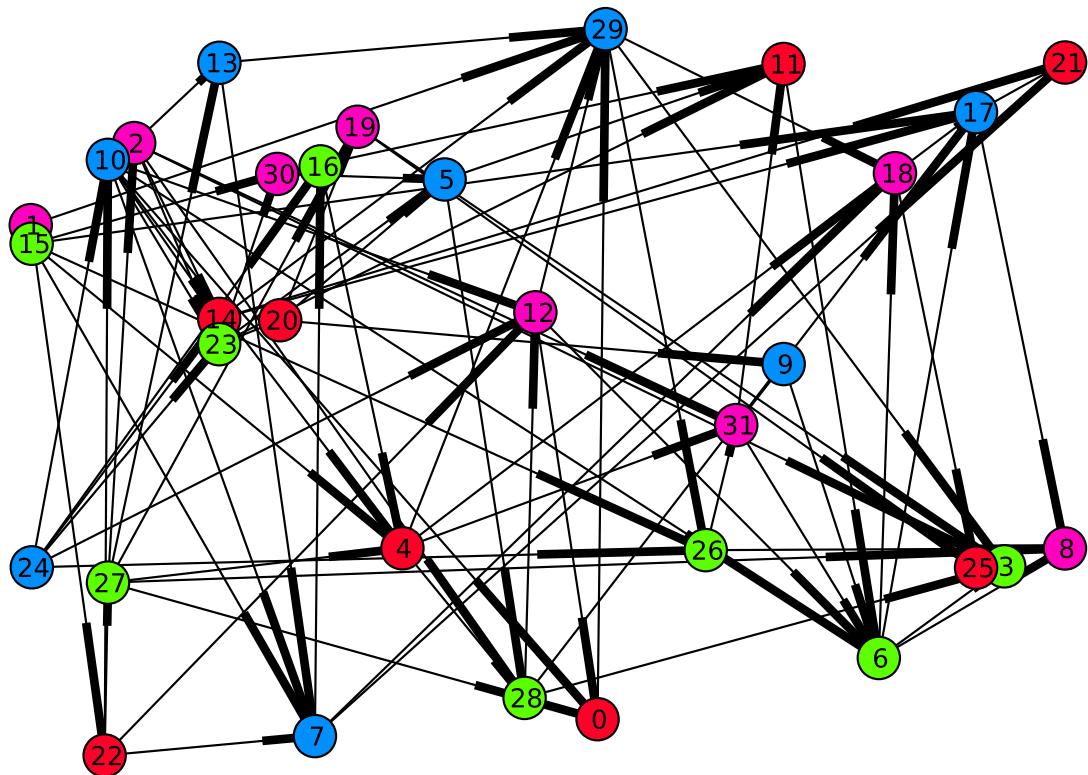


Figure 5.1: Time started on x, runtime on y, and color-coded by engine-id (in this case there were four engines). Edges denote dependencies.

5.11 Details of Parallel Computing with IPython

Note: There are still many sections to fill out in this doc

5.11.1 Caveats

First, some caveats about the detailed workings of parallel computing with 0MQ 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((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 echo
```

```
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')
```

5.11.2 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, which is used by all higher level methods of its Views.

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:

returnsAsyncResult

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 `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: Examples for execute and run

5.11.3 Views

The principal extension of the Client is the View class. The client is typically a singleton for connecting to a cluster, and presents a low-level interface to the Hub and Engines. Most real usage will involve creating one or more View objects for working with engines in various ways.

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).

For instance, to get the process ID of all your engines:

```
In [5]: import os

In [6]: dview.apply_sync(os.getpid)
Out[6]: [1354, 1356, 1358, 1360]
```

Or to see the hostname of the machine they are on:

```
In [5]: import socket  
  
In [6]: dview.apply_sync(socket.gethostname)  
Out[6]: ['tesla', 'tesla', 'edison', 'edison', 'edison']
```

Note: TODO: expand on direct execution

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()
```

Note: TODO: write this section

LoadBalancedView

The `LoadBalancedView` is the class for load-balanced execution via the task scheduler. These views always run tasks on exactly one engine, but let the scheduler determine where that should be, allowing load-balancing of tasks. The `LoadBalancedView` does allow you to specify restrictions on where and when tasks can execute, for more complicated load-balanced workflows.

5.11.4 Data Movement

Since the `LoadBalancedView` does not know where execution will take place, explicit data movement methods like push/pull and scatter/gather do not make sense, and are not provided.

5.11.5 Results

AsyncResult

Our primary representation of the results of remote execution 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 `AsyncResult`s 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 0MQ 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 `AsyncResult`s.

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

5.11.6 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)

5.11.7 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.

5.11.8 Synchronization

Since the Client is a synchronous object, events do not automatically trigger in your interactive session - you must poll the 0MQ 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

5.11.9 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 `map()`, which does exactly this, and many parallel execution tools in Python, such as the built-in `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`

5.11.10 Decorators and RemoteFunctions

Note: TODO: write this section

```
@parallel()  
@remote()  
RemoteFunction  
ParallelFunction
```

5.11.11 Dependencies

Note: TODO: write this section

```
@depend()  
@require()  
Dependency
```

5.12 Transitioning from IPython.kernel to IPython.parallel

We have rewritten our parallel computing tools to use [ZeroMQ](#) 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.

5.12.1 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 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.

5.12.2 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()
```

5.12.3 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.

5.12.4 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.

5.12.5 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 users how to configure IPython to their liking. Developers 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 *one or more* configuration files and a single set of 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 configurable objects implement the actual logic of the application and know how to configure themselves given the configuration object.

Applications always have a `log` attribute that is a configured Logger. This allows centralized logging configuration per-application.

Configurable: `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 thing.

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.

Singletons: `SingletonConfigurable` Any object for which there is a single canonical instance. These are just like Configurables, except they have a class method `instance()`, that returns the current active instance (or creates one if it does not exist). Examples of singletons include `InteractiveShell`. This lets objects easily connect to the current running Application without passing objects around everywhere. For instance, to get the current running Application instance, simply do: `app = Application.instance()`.

Note: Singletons are not strictly enforced - you can have many instances of a given singleton class, but the `instance()` method will always return the same one.

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 configurable subclass:

```
# Sample configurable:
from IPython.config.configurable import Configurable
from IPython.utils.traitlets import Int, Float, Unicode, Bool

class MyClass(Configurable):
    name = Unicode(u'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 `Unicode`, `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.

You can also load configuration files by profile, for instance:

```
load_subconfig('ipython_config.py', profile='default')
```

to inherit your default configuration as a starting point.

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, Unicode, Bool

class Foo(Configurable):
    name = Unicode(u'fooname', config=True)
```

```
value = Float(100.0, config=True)

class Bar(Foo):
    name = Unicode(u'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 = u'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? IPython uses “profiles” for configuration, and by default, all profiles will be stored 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 which profile you are using. For users with a single configuration, this will simply be ‘default’, and will be located in `<IPYTHON_DIR>/profile_default`.

The next thing you need to know is what to call your 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`, and *all* IPython applications will use this file. Other applications, such as the

parallel **ipcluster** scripts or the QtConsole will load their own config files *after* `ipython_config.py`. To load a particular configuration file instead of the default, the name can be overridden by the `config_file` command line flag.

To generate the default configuration files, do:

```
$> ipython profile create
```

and you will have a default `ipython_config.py` in your IPython directory under `profile_default`. If you want the default config files for the `IPython.parallel` applications, add `--parallel` to the end of the command-line args.

6.1.6 Profiles

A profile is a directory containing configuration and runtime files, such as logs, connection info for the parallel apps, and your IPython command history.

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 files, logs, and histories for each of these purposes.

Let's start by showing how a profile is used:

```
$ ipython --profile=sympy
```

This tells the **ipython** command line program to get its configuration from the “sympy” profile. The file names for various profiles do not change. The only difference is that profiles are named in a special way. In the case above, the “sympy” profile means looking for `ipython_config.py` in `<IPYTHON_DIR>/profile_sympy`.

The general pattern is this: simply create a new profile with:

```
ipython profile create <name>
```

which adds a directory called `profile_<name>` to your IPython directory. Then you can load this profile by adding `--profile=<name>` to your command line options. Profiles are supported by all IPython applications.

IPython ships with some sample profiles in `IPython/config/profile`. If you create profiles with the name of one of our shipped profiles, these config files will be copied over instead of starting with the automatically generated config files.

6.1.7 Command-line arguments

IPython exposes *all* configurable options on the command-line. The command-line arguments are generated from the Configurable traits of the classes associated with a given Application. Configuring IPython from the command-line may look very similar to an IPython config file

IPython applications use a parser called `KeyValueLoader` to load values into a Config object. Values are assigned in much the same way as in a config file:

```
$> ipython --InteractiveShell.use_readline=False --BaseIPythonApplication.profile='myprofile'
```

Is the same as adding:

```
c.InteractiveShell.use_readline=False  
c.BaseIPythonApplication.profile='myprofile'
```

to your config file. Key/Value arguments *always* take a value, separated by ‘=’ and no spaces.

Aliases

For convenience, applications have a mapping of commonly used traits, so you don’t have to specify the whole class name. For these **aliases**, the class need not be specified:

```
$> ipython --profile='myprofile'  
# is equivalent to  
$> ipython --BaseIPythonApplication.profile='myprofile'
```

Flags

Applications can also be passed **flags**. Flags are options that take no arguments, and are always prefixed with `--`. They are simply wrappers for setting one or more configurables with predefined values, often True/False.

For instance:

```
$> ipcontroller --debug  
# is equivalent to  
$> ipcontroller --Application.log_level=DEBUG  
# and  
$> ipython --pylab  
# is equivalent to  
$> ipython --pylab=auto
```

Subcommands

Some IPython applications have **subcommands**. Subcommands are modeled after **git**, and are called with the form **command subcommand [...args]**. Currently, the QtConsole is a subcommand of terminal IPython:

```
$> ipython qtconsole --profile=myprofile
```

and **ipcluster** is simply a wrapper for its various subcommands (start, stop, engines).

```
$> ipcluster start --profile=myprofile --n=4
```

To see a list of the available aliases, flags, and subcommands for an IPython application, simply pass `-h` or `--help`. And to see the full list of configurable options (*very long*), pass `--help-all`.

6.1.8 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. Fourth, 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 functions. By defining these functions, users can customize IPython by accessing the actual runtime objects of IPython. Here is a sample extension:

```
# myextension.py

def load_ipython_extension(ipython):
    # The ``ipython`` argument is the currently active
    # :class:`InteractiveShell` instance that can be used in any way.
    # This allows you do to things like register new magics, plugins or
    # aliases.

def unload_ipython_extension(ipython):
    # If you want your extension to be unloadable, put that logic here.
```

This `load_ipython_extension()` function is called after your extension is imported and the currently active `InteractiveShell` instance is passed as the only argument. You can do anything you want with IPython at that point.

The `load_ipython_extension()` will be called again if you load or reload the extension again. It is up to the extension author to add code to manage that.

You can put your extension modules anywhere you want, as long as they can be imported by Python's standard import mechanism. However, to make it easy to write extensions, you can also put your extensions in `os.path.join(self.ipython_dir, 'extensions')`. This directory is added to `sys.path` automatically.

6.2.1 Using extensions

There are two ways you can tell IPython to use your extension:

1. Listing it in a configuration file.
2. Using the `%load_ext` magic function.

To load an extension called `myextension.py` add the following logic to your configuration file:

```
c.InteractiveShellApp.extensions = [  
    'myextension'  
]
```

To load that same extension at runtime, use the `%load_ext` magic:

```
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 traited 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 [`Configurable`](#).

Applications themselves are `Configurable` as well, so we will start with some application-level config.

6.4.1 Application-level configuration

Assuming that your configuration file has the following at the top:

```
c = get_config()
```

the following attributes are set application-wide:

terminal IPython-only flags:

c.TerminalIPythonApp.display_banner A boolean that determined if the banner is printed when **ipython** is started.

c.TerminalIPythonApp.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.TerminalIPythonApp.nosep A boolean that determines if there should be no blank lines between prompts.

c.Application.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. This can also be set by the name of the logging level, e.g. INFO=20, WARN=30.

Some options, such as extensions and startup code, can be set for any application that starts an [`InteractiveShell`](#). These apps are subclasses of [`InteractiveShellApp`](#). Since subclasses inherit configuration, setting a trait of `c.InteractiveShellApp` will affect all IPython applications, but if you want terminal IPython and the QtConsole to have different values, you can set them via `c.TerminalIPythonApp` and `c.IPKernelApp` respectively.

c.InteractiveShellApp.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. `InteractiveShellApp.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. `InteractiveShellApp.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.IPythonTerminalApp.display_banner = True
c.InteractiveShellApp.log_level = 20
c.InteractiveShellApp.extensions = [
    'myextension'
]
c.InteractiveShellApp.exec_lines = [
    'import numpy',
    'import scipy'
]
c.InteractiveShellApp.exec_files = [
    'mycode.py',
    'fancy.ipy'
]
c.InteractiveShell.autoindent = True
c.InteractiveShell.colors = 'LightBG'
c.InteractiveShell.confirm_exit = False
c.InteractiveShell.deep_reload = True
```

```
c.InteractiveShell.editor = 'nano'  
c.InteractiveShell.prompt_in1 = 'In [\#]: '  
c.InteractiveShell.prompt_in2 = '... .\D.: '  
c.InteractiveShell.prompt_out = 'Out[\#]: '  
c.InteractiveShell.prompts_pad_left = True  
c.InteractiveShell.xmode = 'Context'  
  
c.PrefilterManager.multi_line_specials = True  
  
c.AliasManager.user_aliases = [  
    ('la', 'ls -al')  
]
```

6.5 Editor configuration

IPython can integrate with text editors in a number of different ways:

- Editors (such as (X)Emacs [Emacs], vim [vim] and TextMate [TextMate]) can send code to IPython for execution.
- IPython's %edit magic command can open an editor of choice to edit a code block.

The %edit command (and its alias %ed) will invoke the editor set in your environment as EDITOR. If this variable is not set, it will default to vi under Linux/Unix and to notepad under Windows. You may want to set this variable properly and to a lightweight editor which doesn't take too long to start (that is, something other than a new instance of Emacs). This way you can edit multi-line code quickly and with the power of a real editor right inside IPython.

You can also control the editor via the command-line option ‘-editor’ or in your configuration file, by setting the `InteractiveShell.editor` configuration attribute.

6.5.1 TextMate

Currently, TextMate support in IPython is broken. It used to work well, but the code has been moved to `IPython.quarantine` until it is updated.

6.5.2 vim configuration

Currently, vim support in IPython is broken. Like the TextMate code, the vim support code has been moved to `IPython.quarantine` until it is updated.

6.5.3 (X)Emacs

6.5.4 Editor

If you are a dedicated Emacs user, and want to use Emacs when IPython’s %edit magic command is called you should set up the Emacs server so that new requests are handled by the original process. This means that

almost no time is spent in handling the request (assuming an Emacs process is already running). For this to work, you need to set your EDITOR environment variable to ‘emacsclient’. The code below, supplied by Francois Pinard, can then be used in your `.emacs` file to enable the server:

```
(defvar server-buffer-clients)
(when (and (fboundp 'server-start) (string-equal (getenv "TERM") 'xterm))
  (server-start)
  (defun fp-kill-server-with-buffer-routine ()
    (and server-buffer-clients (server-done)))
  (add-hook 'kill-buffer-hook 'fp-kill-server-with-buffer-routine))
```

Thanks to the work of Alexander Schmolck and Prabhu Ramachandran, currently (X)Emacs and IPython get along very well in other ways.

Note: You will need to use a recent enough version of `python-mode.el`, along with the file `ipython.el`. You can check that the version you have of `python-mode.el` is new enough by either looking at the revision number in the file itself, or asking for it in (X)Emacs via `M-x py-version`. Versions 4.68 and newer contain the necessary fixes for proper IPython support.

The file `ipython.el` is included with the IPython distribution, in the directory `docs/emacs`. Once you put these files in your Emacs path, all you need in your `.emacs` file is:

```
(require 'ipython)
```

This should give you full support for executing code snippets via IPython, opening IPython as your Python shell via `C-c !`, etc.

You can customize the arguments passed to the IPython instance at startup by setting the `py-python-command-args` variable. For example, to start always in `pylab` mode with hardcoded light-background colors, you can use:

```
(setq py-python-command-args '("-pylab" "-colors" "LightBG"))
```

If you happen to get garbage instead of colored prompts as described in the previous section, you may need to set also in your `.emacs` file:

```
(setq ansi-color-for-comint-mode t)
```

Notes on emacs support:

- There is one caveat you should be aware of: you must start the IPython shell before attempting to execute any code regions via `C-c |`. Simply type `C-c !` to start IPython before passing any code regions to the interpreter, and you shouldn’t experience any problems. This is due to a bug in Python itself, which has been fixed for Python 2.3, but exists as of Python 2.2.2 (reported as SF bug [737947]).
- The (X)Emacs support is maintained by Alexander Schmolck, so all comments/requests should be directed to him through the IPython mailing lists.
- This code is still somewhat experimental so it’s a bit rough around the edges (although in practice, it works quite well).

- Be aware that if you customized `py-python-command` previously, this value will override what `ipython.el` does (because loading the customization variables comes later).

6.6 Outdated configuration information that might still be useful

Warning: All of the information in this file is outdated. Until the new configuration system is better documented, this material is being kept.

This section will help you set various things in your environment for your IPython sessions to be as efficient as possible. All of IPython's configuration information, along with several example files, is stored in a directory named by default `$HOME/.config/ipython` if `$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 pagers (such as Windows), IPython will use a very limited builtin pager.

6.6.4 Fine-tuning your prompt

IPython's prompts can be customized using a syntax similar to that of the bash shell. Many of bash's escapes are supported, as well as a few additional ones. We list them below:

```
\#
    the prompt/history count number. This escape is automatically
    wrapped in the coloring codes for the currently active color scheme.
\N
    the 'naked' prompt/history count number: this is just the number
    itself, without any coloring applied to it. This lets you produce
    numbered prompts with your own colors.
\D
    the prompt/history count, with the actual digits replaced by dots.
    Used mainly in continuation prompts (prompt_in2)
\w
    the current working directory
\W
    the basename of current working directory
\Xn
    where $n=0\ldots5. The current working directory, with $HOME
    replaced by ~, and filtered out to contain only $n$ path elements
\Yn
    Similar to \Xn, but with the $n+1$ element included if it is ~ (this
    is similar to the behavior of the %n 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
\vv
    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”)}’ 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 <who@somewhere.com>"
```

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.

7.1.3 Commit messages

Good commit messages are very important; they provide a verbal account of what happened that is often invaluable for anyone trying to understand the intent of a commit later on (including the original author!). And git's log command is a very versatile and powerful tool, capable of extracting a lot of information from the commit logs, so it's important that these logs actually have useful information in them.

In short, a commit message should have the form:

One line summary.

<THIS LINE MUST BE LEFT BLANK>

More detailed description of what was done, using multiple lines and even more than one paragraph if needed. For very simple commits this may not be necessary, but non-trivial ones should always have it.

Closes gh-NNN. # if the commit closes issue NNN on github.

This format is understood by many git tools that expect a *single line* summary, so please do respect it.

An excellent reference on commits message is [this blog post](#), please take a moment to read it (it's short but very informative).

7.2 Working with IPython source code

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](#).

Contents:

7.2.1 Install git

Overview

| | |
|-----------------|--|
| Debian / Ubuntu | <code>sudo apt-get install git-core</code> |
| Fedora | <code>sudo yum install git-core</code> |
| Windows | Download and install msysGit |
| OS X | Use the git-osx-installer |

In detail

See the [git](#) page for the most recent information.

Have a look at the [github](#) install help pages available from [github help](#)

There are good instructions here: http://book.git-scm.com/2_installing_git.html

7.2.2 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.

7.2.3 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?](#).

- When you have finished, check you have committed all your changes:

```
git status
```

- 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](#).

7.2.4 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.

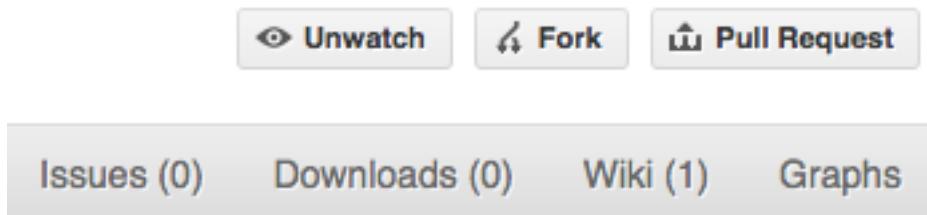
Set up and configure a github account

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`:



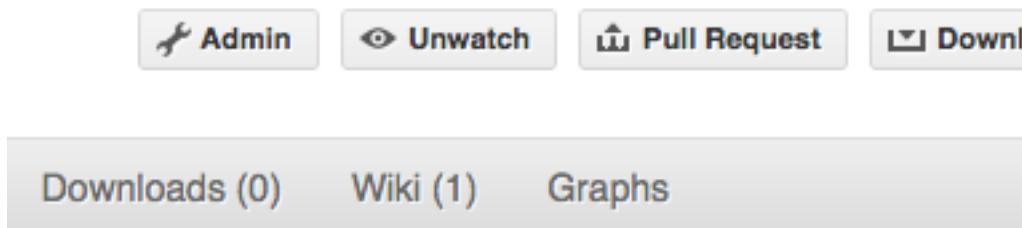
4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the [ipython mailing list](#),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

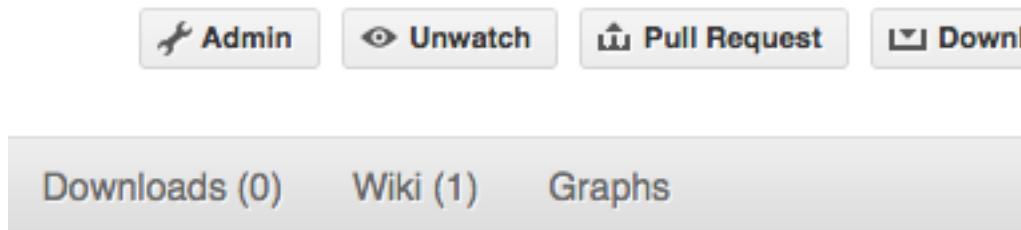
Several people sharing a single repository

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.

7.2.5 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](#)
- The [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 = []

    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) 2011, the 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, similar (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.dtexample` 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 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 write this type of test, where your docstring is meant *only* as a test, by prefixing the function name with `doctest_` and leaving its body *absolutely empty* other than the docstring. In `IPython.core.tests.test_magic` you can find several examples of this, but for completeness sake, your code should look like this (a simple case):

```
def doctest_time():
    """
    In [10]: %time None
    CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
    Wall time: 0.00 s
    """
```

This function is only analyzed for its docstring but it is not considered a separate test, which is why its body should be empty.

Parametric tests done right

If you need to run multiple tests inside the same standalone function or method of a `unittest.TestCase` subclass, IPython provides the `parametric` decorator for this purpose. This is superior to how test generators work in nose, because IPython's keeps intact your stack, which makes debugging vastly easier. For example, these are some parametric tests both in class form and as a standalone function (choose in each situation the style that best fits the problem at hand, since both work):

```
from IPython.testing import decorators as dec

def is_smaller(i,j):
    assert i<j, "%s < %s" % (i,j)

class Tester(ParametricTestCase):

    def test_parametric(self):
        yield is_smaller(3, 4)
        x, y = 1, 2
        yield is_smaller(x, y)

@dec.parametric
def test_par_standalone():
    yield is_smaller(3, 4)
    x, y = 1, 2
    yield is_smaller(x, y)
```

Writing tests for Twisted-using code

Tests of Twisted [Twisted] using code should be written by subclassing the `TestCase` class that comes with `twisted.trial.unittest`. Furthermore, all `Deferred` instances that are created in the test must be properly chained and the final one *must* be the return value of the test method.

Note: The best place to see how to use the testing tools, are the tests for these tools themselves, which live in `IPython.testing.tests`.

7.5.4 Design requirements

This section is a set of notes on the key points of the IPython testing needs, that were used when writing the system and should be kept for reference as it evolves.

Testing IPython in full requires modifications to the default behavior of nose and doctest, because the IPython prompt is not recognized to determine Python input, and because IPython admits user input that is not valid Python (things like %magics and !system commands).

We basically need to be able to test the following types of code:

1. Pure Python files containing normal tests. These are not a problem, since Nose will pick them up as long as they conform to the (flexible) conventions used by nose to recognize tests.
2. Python files containing doctests. Here, we have two possibilities:
 - The prompts are the usual >>> and the input is pure Python.
 - The prompts are of the form In [1] : and the input can contain extended IPython expressions.

In the first case, Nose will recognize the doctests as long as it is called with the --with-doctest flag. But the second case will likely require modifications or the writing of a new doctest plugin for Nose that is IPython-aware.

3. ReStructuredText files that contain code blocks. For this type of file, we have three distinct possibilities for the code blocks:
 - They use >>> prompts.
 - They use In [1] : prompts.
 - They are standalone blocks of pure Python code without any prompts.

The first two cases are similar to the situation #2 above, except that in this case the doctests must be extracted from input code blocks using docutils instead of from the Python docstrings.

In the third case, we must have a convention for distinguishing code blocks that are meant for execution from others that may be snippets of shell code or other examples not meant to be run. One possibility is to assume that all indented code blocks are meant for execution, but to have a special docutils directive for input that should not be executed.

For those code blocks that we will execute, the convention used will simply be that they get called and are considered successful if they run to completion without raising errors. This is similar to what Nose does for standalone test functions, and by putting asserts or other forms of exception-raising statements it becomes possible to have literate examples that double as lightweight tests.

4. Extension modules with doctests in function and method docstrings. Currently Nose simply can't find these docstrings correctly, because the underlying doctest DocTestFinder object fails there. Similarly to #2 above, the docstrings could have either pure python or IPython prompts.

Of these, only 3-c (reST with standalone code blocks) is not implemented at this point.

7.6 Releasing IPython

This section contains notes about the process that is used to release IPython. Our release process is currently not very formal and could be improved.

Most of the release process is automated by the `release` script in the `tools` directory. This is just a handy reminder for the release manager.

1. For writing release notes, this will cleanly show who contributed as author of commits (get the previous release name from the tag list with `git tag`):

```
git log --pretty=format:"* %an" PREV_RELEASE... | sort | uniq
```

2. 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.

3. 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.

4. 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 Launcphad.

5. Drafting a short release announcement with i) highlights and ii) a link to the html version of the [What's new](#) section of the documentation.

6. Make sure that the released version of the docs is live on the site. For this we are now using the gh-pages system:
 - Make a static directory for the final copy of the release docs.
 - Update the `index.rst` file and run `build_index.py` to update the html version.
 - Update the `stable` symlink to point to the released version.
 - Run `git add` for all the new files and commit.
 - Run `git push` to update the public version of the docs on gh-pages.

7. 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 configurables (mostly done).

Release 0.12

- Continue to refactor IPython's core by turning everything into configurables.
- Merge draft html notebook (started).
- Add two-process Terminal frontend using ZMQ architecture.

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 configurable class for this is `IPython.core.configurable.Configurable`. This section outlines the status of this work.

Parts of the IPython core that have been turned into configurables:

- The main `InteractiveShell` class.
- The aliases (`IPython.core.alias`).
- History management (`IPython.core.history`).
- Plugins (`IPython.core.plugin`).
- The display and builtin traps (`IPython.core.display_trap` and `IPython.core.builtin_trap`).
- The prefilter machinery (`IPython.core.prefilter`).

Parts of the IPython core that still need to be Configurable:

- Magics.
- Prompts.
- Tab completers.
- Exception handling.

- Anything else.

Process management for `IPython.parallel`

A few things need to be done to improve how processes are started up and managed for the parallel computing side of IPython:

- Improve the SSH launcher so it is at least back to the levels of 0.10.2
- We need to add support for other batch systems (LSF, Condor, etc.).

7.7.3 Porting to 3.0

Dropping 2.5 support was a major step towards working with Python 3 due to future syntax support in 2.6. IPython 0.11 requires 2.6 now, so 0.10.2 will be the last IPython release that supports Python 2.5.

We currently have a [separate repo](#) tracking IPython development that works with Python 3. The core of IPython does work, but the parallel computing code in `IPython.parallel` does not yet work in Python 3, though the only major dependency of the parallel code, `pymq`, does work on Python 3.

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.

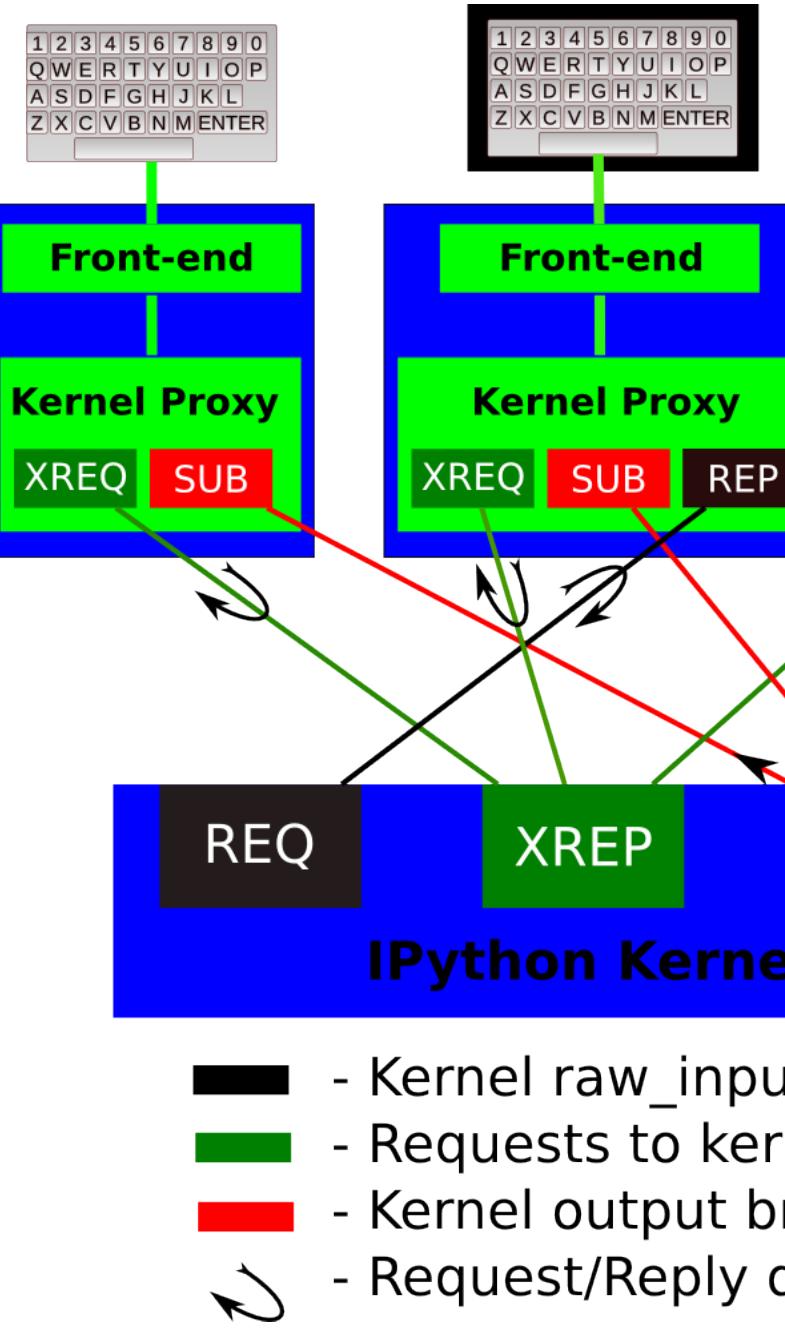
- `IPython.frontend`. This package contains the various IPython frontends. Currently, the code in this subpackage is very experimental and may be broken.
- `IPython.gui`. Another semi-experimental wxPython based IPython GUI.
- `IPython.kernel`. This contains IPython's parallel computing system.
- `IPython.lib`. 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 socket 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 withPython/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,

# The string form of the object, possibly truncated for length if
# detail_level is 0
'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,

    # So far, this can be 'range', 'tail' or 'search'.
    'hist_access_type' : str,

    # If hist_access_type is 'range', get a range of input cells. session can
    # be a positive session number, or a negative number to count back from
    # the current session.
    'session' : int,
    # start and stop are line numbers within that session.
    'start' : int,
    'stop' : int,

    # If hist_access_type is 'tail', get the last n cells.
    'n' : int,

    # If hist_access_type is 'search', get cells matching the specified glob
    # pattern (with * and ? as wildcards).
    'pattern' : str,
}

}
```

Message type: history_reply:

```
content = {
    # A list of 3 tuples, either:
    # (session, line_number, input) or
    # (session, line_number, (input, output)),
    # depending on whether output was False or True, respectively.
    'history' : list,
}
```

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 kids 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 an `sys.displayhook` 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 if 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/O
    '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:  
    'abcd-...' : 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,kwags)
```

`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 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 4 (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.

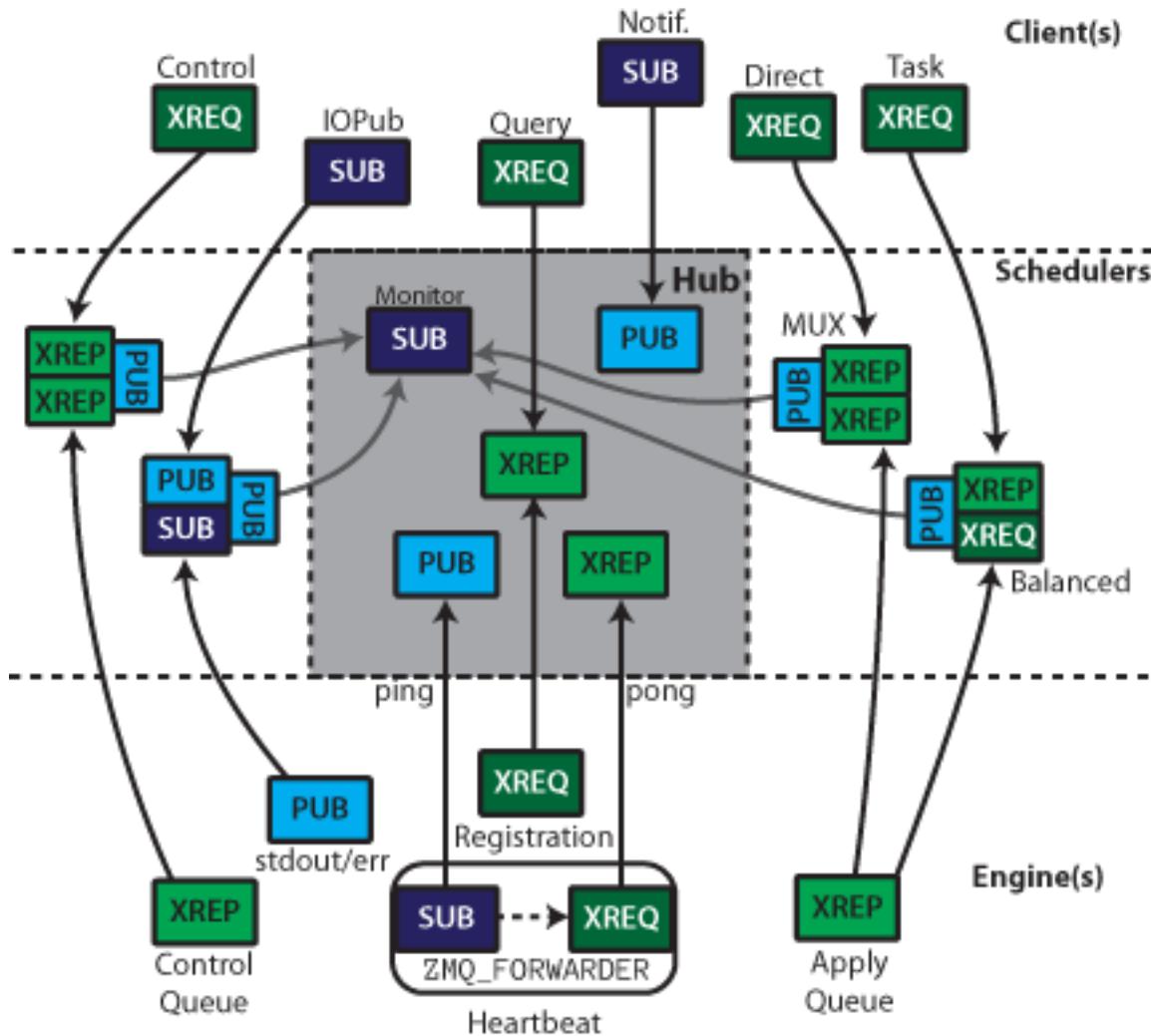


Figure 7.1: All the connections involved in connecting one client to one engine.

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

Registration

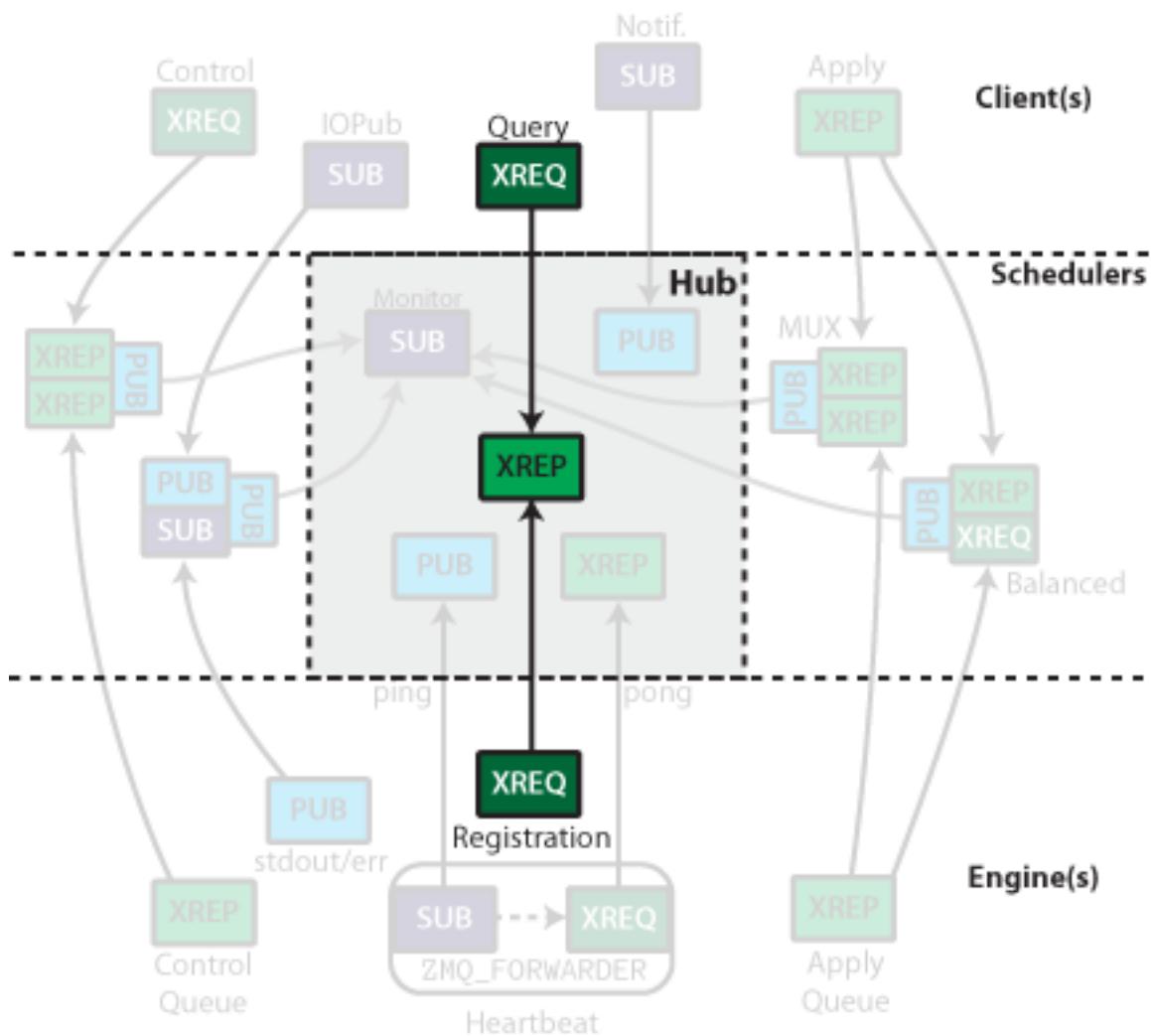


Figure 7.2: Engines and Clients only need to know where the Query XREP is located to start connecting.

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.

Heartbeat

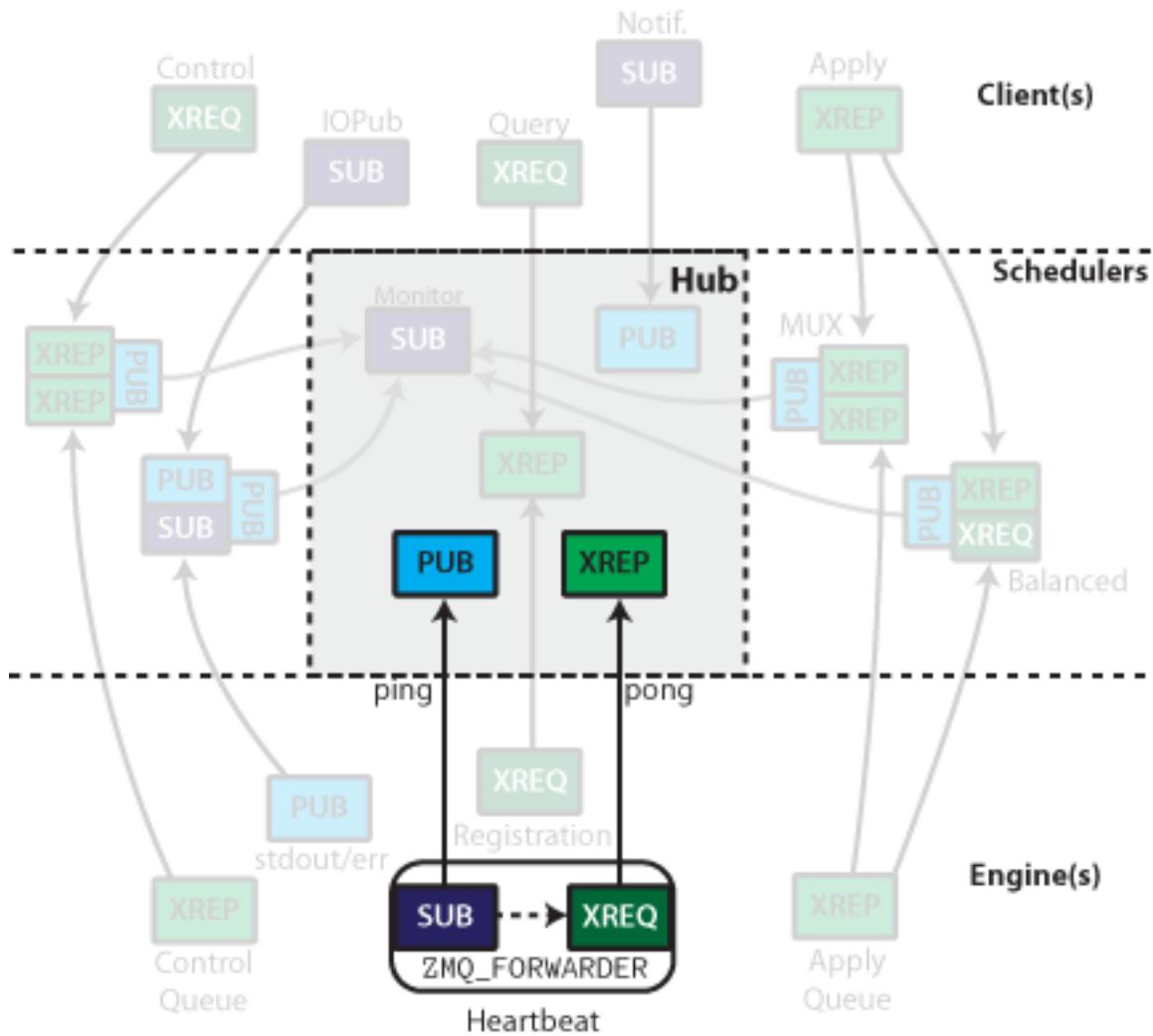


Figure 7.3: The heartbeat sockets.

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

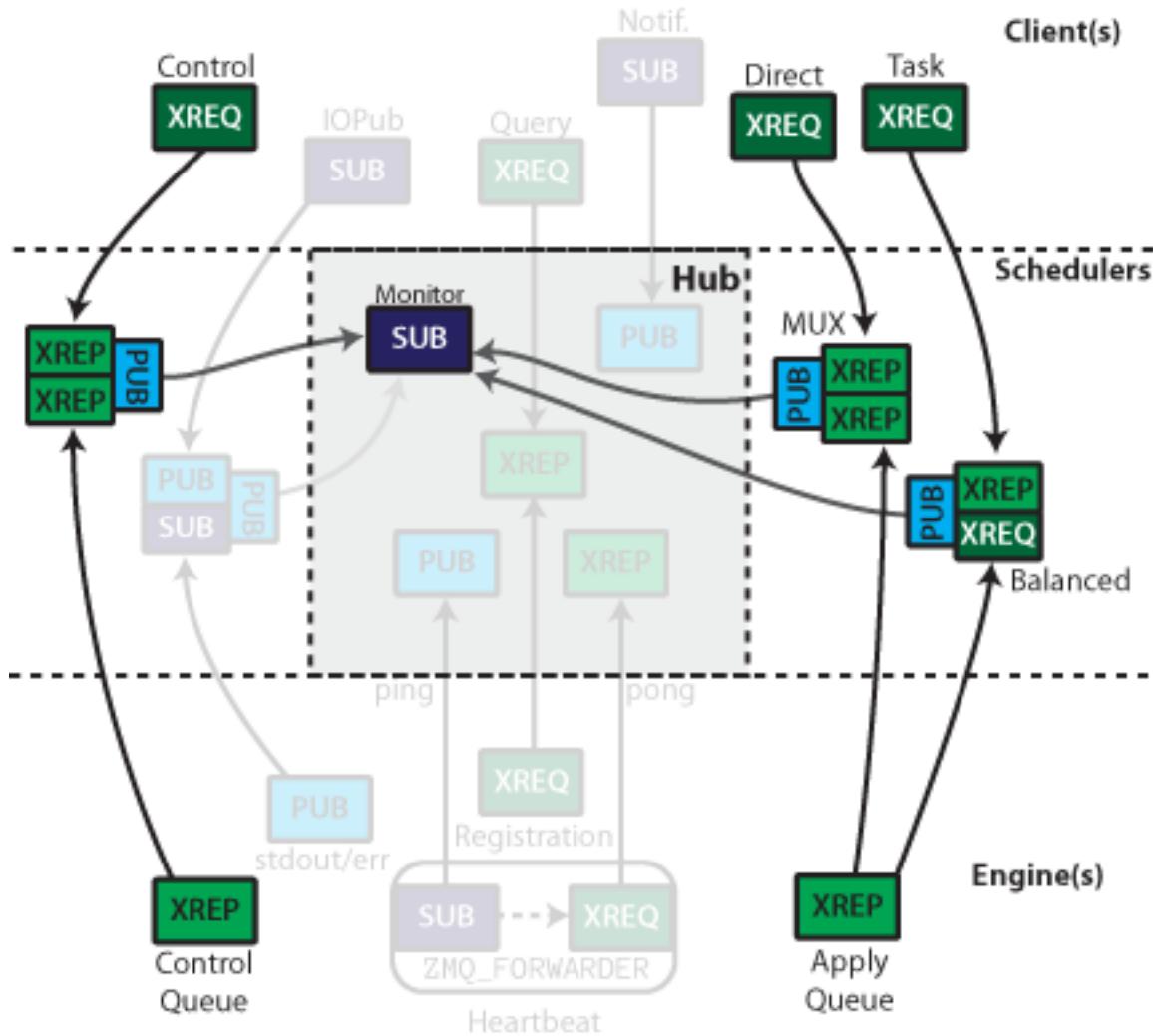


Figure 7.4: Control message scheduler on the left, execution (apply) schedulers on the right.

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

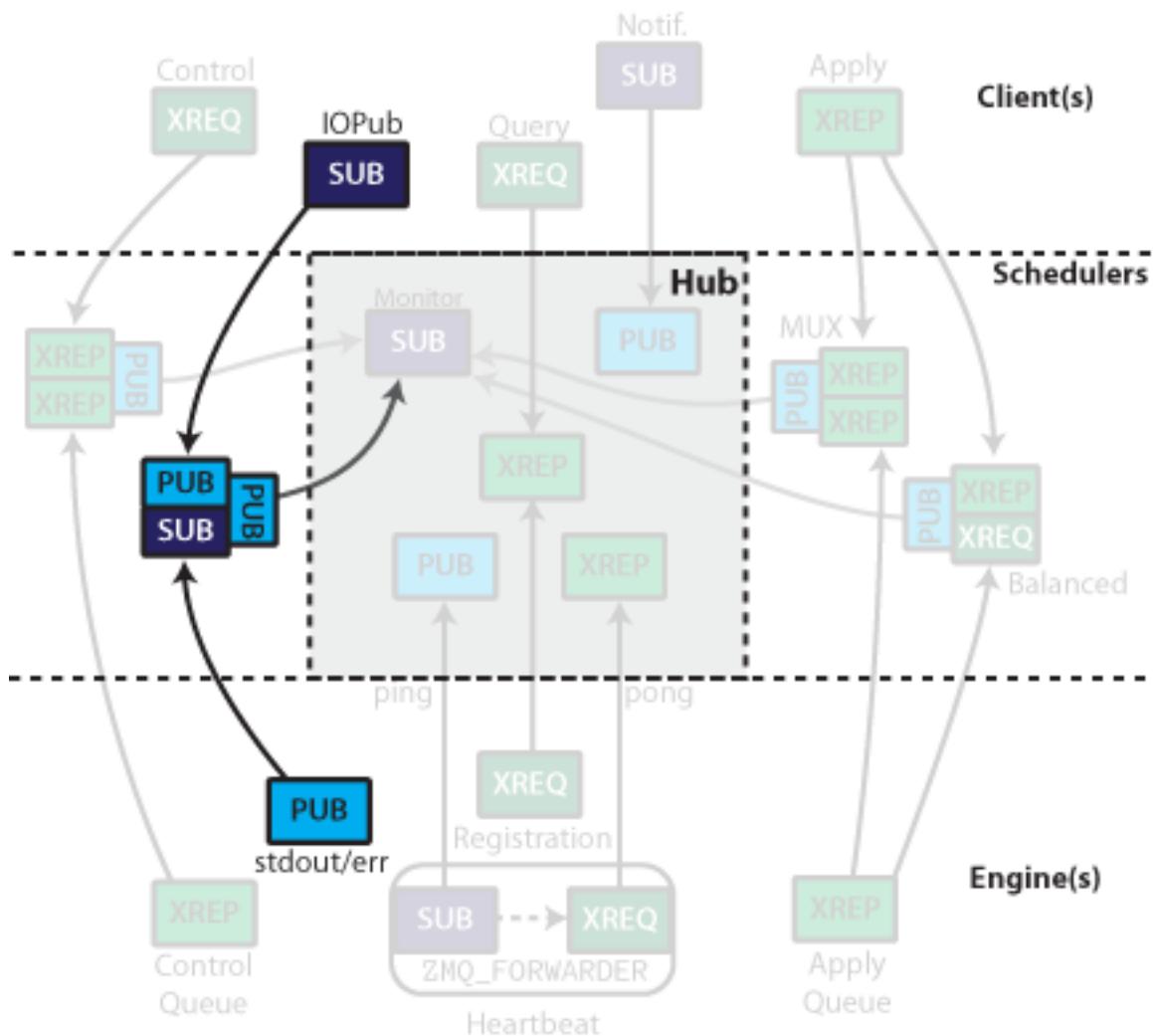


Figure 7.5: stdout/err are published via a PUB / SUB MonitoredQueue

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

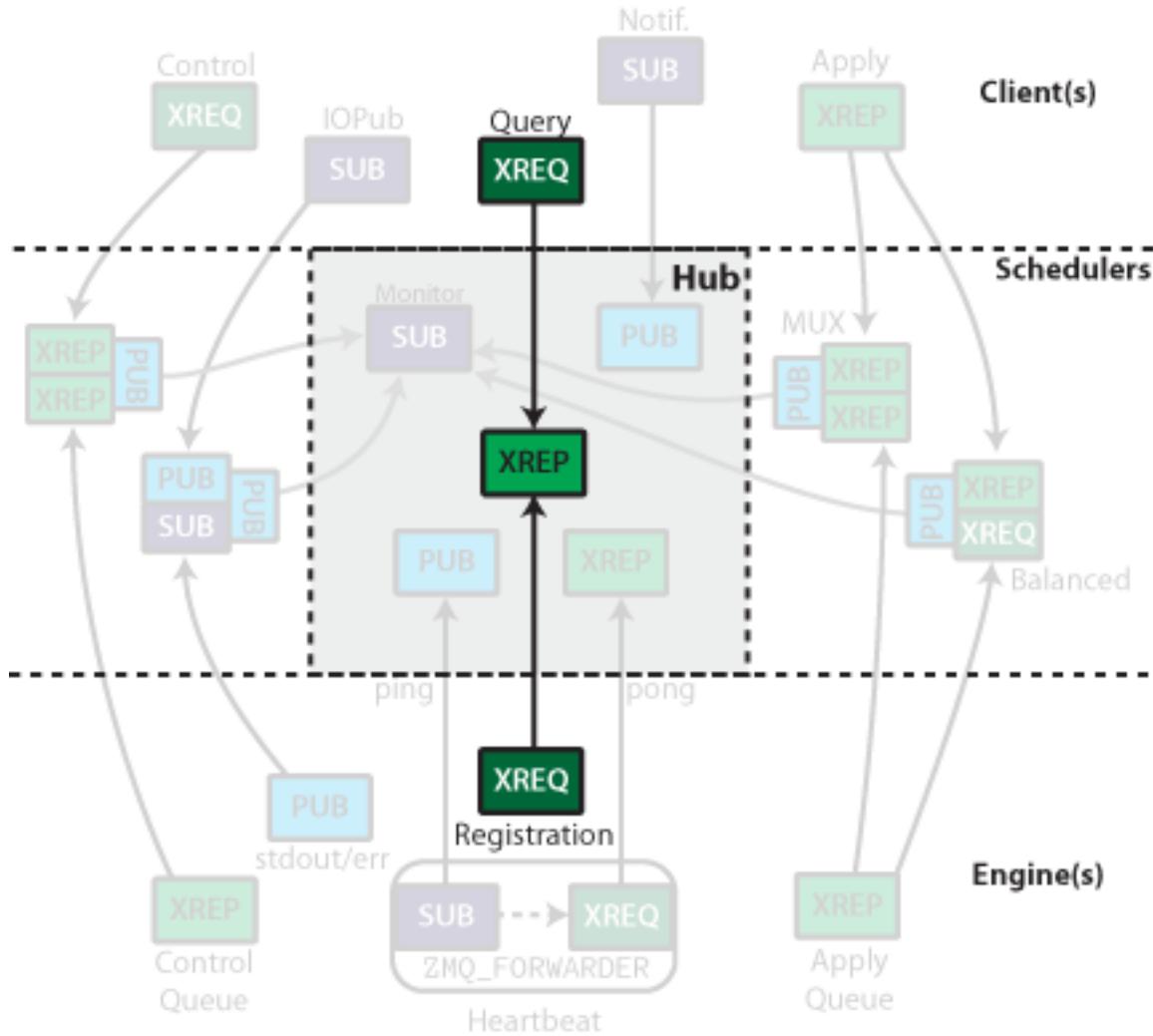


Figure 7.6: Clients connect to an XREP socket to query the hub.

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.

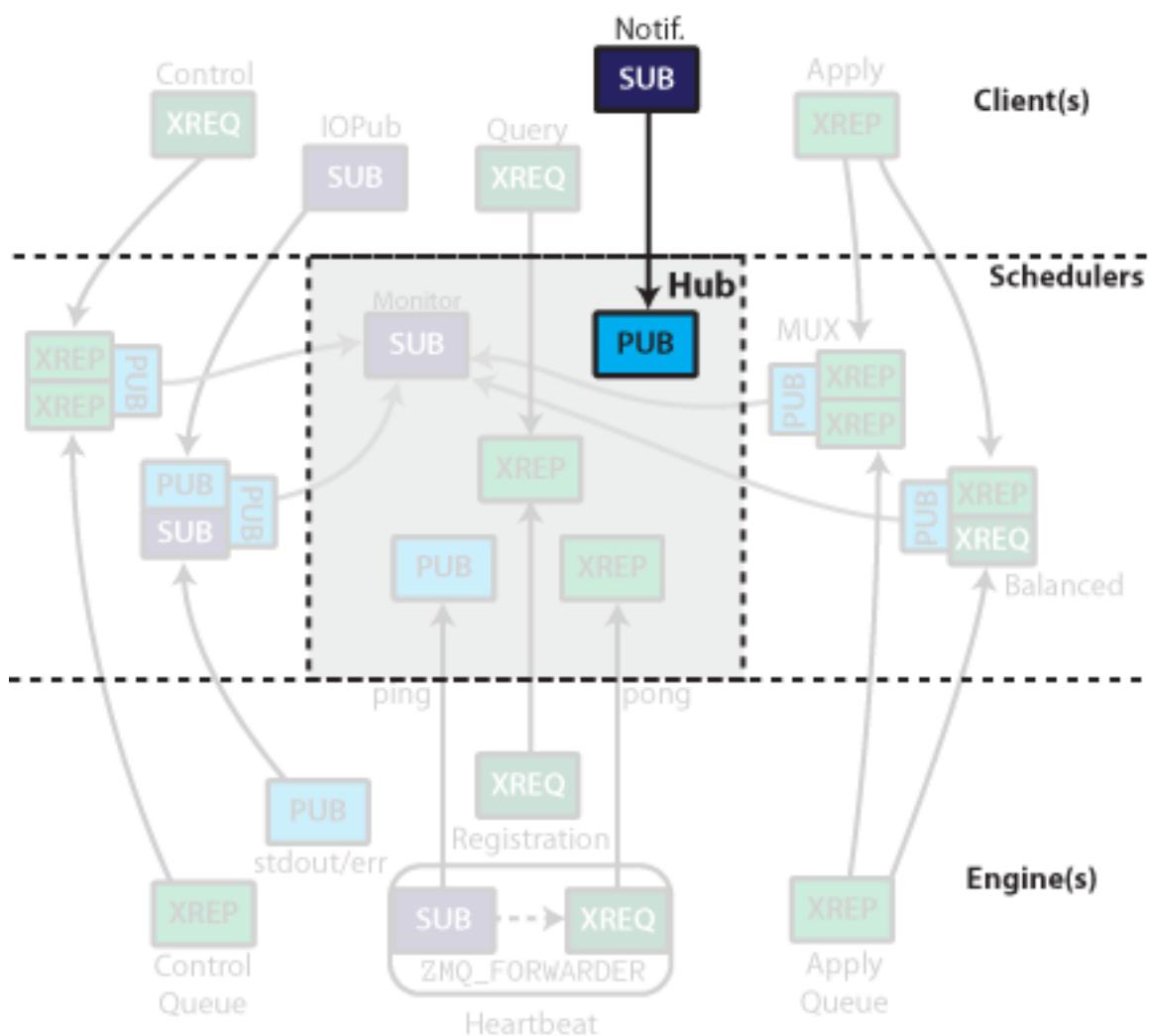


Figure 7.7: Engine registration events are published via a PUB socket.

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 Notes on code execution in InteractiveShell

7.13.1 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.

7.13.2 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()
```

7.14 IPython Qt interface

7.14.1 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.

7.14.2 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.

7.14.3 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.

7.15 Porting IPython to a two process model using zeromq

7.15.1 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.

7.15.2 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.

7.15.3 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/completer/examples/kernel/kernel.py>
- <http://github.com/ellisonbg/pyzmq/blob/completer/examples/kernel/completer.py>
- <http://github.com/fperez/pyzmq/blob/completer/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/completer/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.

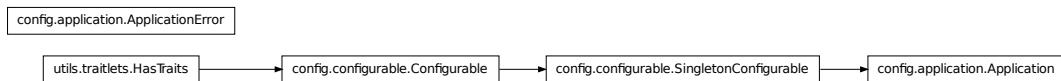
The crash recovery system, is a IPython kernel module for when it fails unexpectedly, you can retrieve the information from the section, this will be based on a log and a lock file to indicate when the kernel was not closed in a proper way.

THE IPYTHON API

8.1 config.application

8.1.1 Module: config.application

Inheritance diagram for IPython.config.application:



A base class for a configurable application.

Authors:

- Brian Granger
- Min RK

8.1.2 Classes

Application

```
class IPython.config.application.Application(**kwargs)
Bases: IPython.config.configurable.SingletonConfigurable
```

A singleton application with full configuration support.

`__init__(**kwargs)`

`aliases`

An instance of a Python dict.

`classmethod class_config_section()`

Get the config class config section

classmethod `class_get_help()`
Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`
Get the help string for a single trait.

classmethod `class_print_help()`
Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes
An instance of a Python list.

classmethod `clear_instance()`
unset _instance for this class and singleton parents.

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.

created = None

description
A trait for unicode strings.

examples
A trait for unicode strings.

exit (exit_status=0)

extra_args
An instance of a Python list.

flags
An instance of a Python dict.

generate_config_file()
generate default config file from Configurables

init_logging()
Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

initialize(argv=None)

Do the basic steps to configure me.

Override in subclasses.

initialize_subcommand(subc, argv=None)

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable): pass
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class `instance`:

```
>>> class Bar(SingletonConfigurable): pass
>>> class Bam(Bar): pass
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

keyvalue_description

A trait for unicode strings.

load_config_file(filename, path=None)

Load a .py based config file by filename and path.

log_level

An enum that whose value must be in a given sequence.

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 unintall it.

option_description

A trait for unicode strings.

parse_command_line (`argv=None`)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help()

Print the flag part of the help.

print_help (`classes=False`)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()

Print the subcommand part of the help.

print_version()

Print the version string.

start()

Start the app mainloop.

Override in subclasses.

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

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.

update_config (config)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

ApplicationError

class IPython.config.application.ApplicationError

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args**message**

8.1.3 Function

IPython.config.application.boolean_flag (name, configurable, set_help='', unset_help='')

Helper for building basic -trait, -no-trait flags.

Parameters name : str

The name of the flag.

configurable : str

The ‘Class.trait’ string of the trait to be set/unset with the flag

set_help : unicode

help string for –name flag

unset_help : unicode

help string for –no-name flag

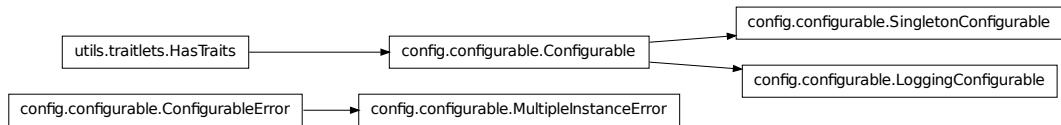
Returns `cfg` : dict

A dict with two keys: ‘name’, and ‘no-name’, for setting and unsetting the trait, respectively.

8.2 config.configurable

8.2.1 Module: config.configurable

Inheritance diagram for IPython.config.configurable:



A base class for objects that are configurable.

Authors:

- Brian Granger
- Fernando Perez
- Min RK

8.2.2 Classes

Configurable

class IPython.config.configurable.**Configurable** (**kwargs)
Bases: IPython.utils.traits.HasTraits

__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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If False (the default), then install the handler. If True then unintall it.

`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.

ConfigurableError

`class IPython.config.configurable.ConfigurableError`

Bases: `exceptions.Exception`

`__init__()`

`x.__init__(...)` initializes x; see `x.__class__.__doc__` for signature

`args`

`message`

LoggingConfigurable

`class IPython.config.configurable.LoggingConfigurable(**kwargs)`

Bases: `IPython.config.configurable.Configurable`

A parent class for Configurables that log.

Subclasses have a log trait, and the default behavior is to get the logger from the currently running Application via Application.instance().log.

`__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.

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

created = None

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.

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.

MultipleInstanceError

class IPython.config.configurable.**MultipleInstanceError**

Bases: IPython.config.configurable.ConfigurableError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

```
args
message
```

SingletonConfigurable

```
class IPython.config.configurable.SingletonConfigurable(**kwargs)
Bases: IPython.config.configurable.Configurable
```

A configurable that only allows one instance.

This class is for classes that should only have one instance of itself or *any* subclass. To create and retrieve such a class use the `SingletonConfigurable.instance()` method.

```
__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.

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(**metadata)`

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(**metadata)`

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classmethod clear_instance()

unset _instance for this class and singleton parents.

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.

created = None

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method create a new instance if none have previously been created and returns a previously created instance is one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable): pass
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable): pass
>>> class Bam(Bar): pass
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

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.

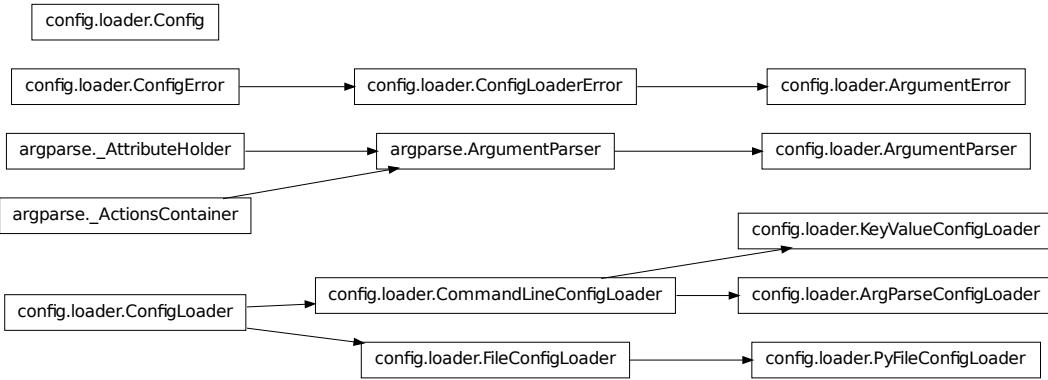
The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.

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.

8.3 config.loader

8.3.1 Module: config.loader

Inheritance diagram for `IPython.config.loader`:



A simple configuration system.

Authors

- Brian Granger
- Fernando Perez
- Min RK

8.3.2 Classes

ArgParseConfigLoader

```
class IPython.config.loader.ArgParseConfigLoader(argv=None, *parser_args,  
                                                **parser_kw)  
Bases: IPython.config.loader.CommandLineConfigLoader
```

A loader that uses the argparse module to load from the command line.

```
__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`.

Returns config : Config

The resulting Config object.

clear()

get_extra_args()

load_config(argv=None)

Parse command line arguments and return as a Config object.

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.

ArgumentError

```
class IPython.config.loader.ArgumentError
Bases: IPython.config.loader.ConfigLoaderError

__init__()
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args
message
```

ArgumentParser

```
class IPython.config.loader.ArgumentParser(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)
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)
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()
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, **kwds)**

Bases: dict

An attribute based dict that can do smart merges.

__init__(*)args, **kwds)**clear**

D.clear() -> None. Remove all items from D.

copy()**static fromkeys(S[, v])** → New dict with keys from S and values equal to v.

v defaults to None.

get

D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.

has_key(key)**items**

D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems

D.iteritems() -> an iterator over the (key, value) items of D

iterkeys

D.iterkeys() -> an iterator over the keys of D

itervalues

D.itervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does:
for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case,
this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

ConfigError

class IPython.config.loader.ConfigError

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

ConfigLoader

class IPython.config.loader.ConfigLoader

Bases: object

A object for loading configurations from just about anywhere.

The resulting configuration is packaged as a Struct.

Notes

A `ConfigLoader` does one thing: load a config from a source (file, command line arguments) and returns the data as a Struct. There are lots of things that `ConfigLoader` does not do. It does not implement complex logic for finding config files. It does not handle default values or merge multiple configs. These things need to be handled elsewhere.

__init__()

A base class for config loaders.

Examples

```
>>> cl = ConfigLoader()  
>>> config = cl.load_config()  
>>> config  
{ }
```

clear()

load_config()

Load a config from somewhere, return a [Config](#) instance.

Usually, this will cause self.config to be set and then returned. However, in most cases, [ConfigLoader.clear\(\)](#) should be called to erase any previous state.

ConfigLoaderError**class IPython.config.loader.ConfigLoaderError**

Bases: [IPython.config.loader.ConfigError](#)

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args**message****FileConfigLoader****class IPython.config.loader.FileConfigLoader**

Bases: [IPython.config.loader.ConfigLoader](#)

A base class for file based configurations.

As we add more file based config loaders, the common logic should go here.

__init__()

A base class for config loaders.

Examples

```
>>> cl = ConfigLoader()  
>>> config = cl.load_config()  
>>> config  
{ }
```

clear()**load_config()**

Load a config from somewhere, return a [Config](#) instance.

Usually, this will cause self.config to be set and then returned. However, in most cases, [ConfigLoader.clear\(\)](#) should be called to erase any previous state.

KeyValueConfigLoader**class IPython.config.loader.KeyValueConfigLoader(argv=None, aliases=None, flags=None)**

Bases: [IPython.config.loader.CommandLineConfigLoader](#)

A config loader that loads key value pairs from the command line.

This allows command line options to be given in the following form:

```
ipython --profile="foo" --InteractiveShell.autocall=False
```

```
__init__(argv=None, aliases=None, flags=None)
```

Create a key value pair config loader.

Parameters `argv` : list

A list that has the form of sys.argv[1:] which has unicode elements of the form u"key=value". If this is None (default), then sys.argv[1:] will be used.

`aliases` : dict

A dict of aliases for configurable traits. Keys are the short aliases, Values are the resolved trait. Of the form: {‘alias’ : ‘Configurable.trait’}

`flags` : dict

A dict of flags, keyed by str name. Values can be Config objects, dicts, or “key=value” strings. If Config or dict, when the flag is triggered, The flag is loaded as `self.config.update(m)`.

Returns `config` : Config

The resulting Config object.

Examples

```
>>> from IPython.config.loader import KeyValueConfigLoader
>>> cl = KeyValueConfigLoader()
>>> cl.load_config(["--A.name='brian'", "--B.number=0"])
{'A': {'name': 'brian'}, 'B': {'number': 0}}
```

```
clear()
```

```
load_config(argv=None, aliases=None, flags=None)
```

Parse the configuration and generate the Config object.

After loading, any arguments that are not key-value or flags will be stored in `self.extra_args` - a list of unparsed command-line arguments. This is used for arguments such as input files or subcommands.

Parameters `argv` : list, optional

A list that has the form of sys.argv[1:] which has unicode elements of the form u"key=value". If this is None (default), then self.argv will be used.

`aliases` : dict

A dict of aliases for configurable traits. Keys are the short aliases, Values are the resolved trait. Of the form: {‘alias’ : ‘Configurable.trait’}

`flags` : dict

A dict of flags, keyed by str name. Values can be Config objects or dicts.
When the flag is triggered, The config is loaded as `self.config.update(cfg)`.

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()

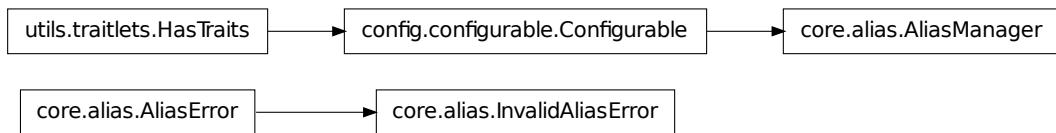
load_config()

Load the config from a file and return it as a Struct.

8.4 core.alias

8.4.1 Module: core.alias

Inheritance diagram for `IPython.core.alias`:



System command aliases.

Authors:

- Fernando Perez
- Brian Granger

8.4.2 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.

    classmethod class_config_section()
        Get the config class config section

    classmethod class_get_help()
        Get the help string for this class in ReST format.

    classmethod class_get_trait_help(trait)
        Get the help string for a single trait.

    classmethod class_print_help()
        Get the help string for a single trait and print it.

    classmethod class_trait_names(**metadata)
        Get a list of all the names of this classes traits.

        This method is just like the trait_names() method, but is unbound.

    classmethod class_traits(**metadata)
        Get a list of all the traits of this class.

        This method is just like the traits() method, but is unbound.

    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.
```

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.

created = None

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]l16> _ip.expand_aliases("np myfile.txt") <16> 'q:/opt/np/notepad++.exe myfile.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 unintall 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

8.4.3 Function

`IPython.core.alias.default_aliases()`

Return list of shell aliases to auto-define.

8.5 core.application

8.5.1 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 components.

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
- Min RK

8.5.2 BaseIPythonApplication

`class IPython.core.application.BaseIPythonApplication(**kwargs)`

Bases: `IPython.config.application.Application`

`__init__(**kwargs)`

`aliases`

An instance of a Python dict.

`auto_create`

A boolean (True, False) trait.

`builtin_profile_dir`

A trait for unicode strings.

`classmethod class_config_section()`

Get the config class config section

classmethod `class_get_help()`
Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`
Get the help string for a single trait.

classmethod `class_print_help()`
Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes
An instance of a Python list.

classmethod `clear_instance()`
unset _instance for this class and singleton parents.

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.

config_file_name
A trait for unicode strings.

config_file_paths
An instance of a Python list.

config_file_specified
A boolean (True, False) trait.

config_files
An instance of a Python list.

copy_config_files
A boolean (True, False) trait.

crash_handler_class
A trait whose value must be a subclass of a specified class.

created = None

description

A trait for unicode strings.

examples

A trait for unicode strings.

exit (exit_status=0)**extra_args**

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

init_config_files()

[optionally] copy default config files into profile dir.

init_crash_handler()

Create a crash handler, typically setting sys.excepthook to it.

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

init_profile_dir()

initialize the profile dir

initialize(argv=None)**initialize_subcommand(subc, argv=None)**

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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_level`

An enum that whose value must be in a given sequence.

`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 unintall it.

`option_description`

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line (argv=None)

Parse the command line arguments.

print_alias_help ()

Print the alias part of the help.

print_description ()

Print the application description.

print_examples ()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help ()

Print the flag part of the help.

print_help (classes=False)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options ()**print_subcommands ()**

Print the subcommand part of the help.

print_version ()

Print the version string.

profile

A trait for unicode strings.

stage_default_config_file ()

auto generate default config file, and stage it into the profile.

start ()

Start the app mainloop.

Override in subclasses.

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

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.

update_config (*config*)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

8.6 core.autocall

8.6.1 Module: core.autocall

Inheritance diagram for IPython.core.autocall:



Autocall capabilities for IPython.core.

Authors:

- Brian Granger
- Fernando Perez
- Thomas Kluyver

Notes

8.6.2 Classes

ExitAutocall

class IPython.core.autocall.**ExitAutocall** (*ip=None*)

Bases: IPython.core.autocall.IPyAutocall

An autocallable object which will be added to the user namespace so that exit, exit(), quit or quit() are all valid ways to close the shell.

```
__init__(ip=None)
rewrite = False
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.
```

IPyAutocall

```
class IPython.core.autocall.IPyAutocall(ip=None)
Bases: object

Instances of this class are always autocalled

This happens regardless of ‘autocall’ variable state. Use this to develop macro-like mechanisms.
```

```
__init__(ip=None)
rewrite = True
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.
```

ZMQExitAutocall

```
class IPython.core.autocall.ZMQExitAutocall(ip=None)
Bases: IPython.core.autocall.ExitAutocall

Exit IPython. Autocallable, so it needn't be explicitly called.
```

Parameters `keep_kernel` : bool
If True, leave the kernel alive. Otherwise, tell the kernel to exit too (default).

```
__init__(ip=None)
rewrite = False
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.
```

8.7 core.builtin_trap

8.7.1 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

8.7.2 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.

    classmethod class_config_section()
        Get the config class config section

    classmethod class_get_help()
        Get the help string for this class in ReST format.

    classmethod class_get_trait_help(trait)
        Get the help string for a single trait.

    classmethod class_print_help()
        Get the help string for a single trait and print it.

    classmethod class_trait_names(**metadata)
        Get a list of all the names of this classes traits.
```

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`

`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 unintall 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.

8.8 core.compilerop

8.8.1 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

8.8.2 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*)

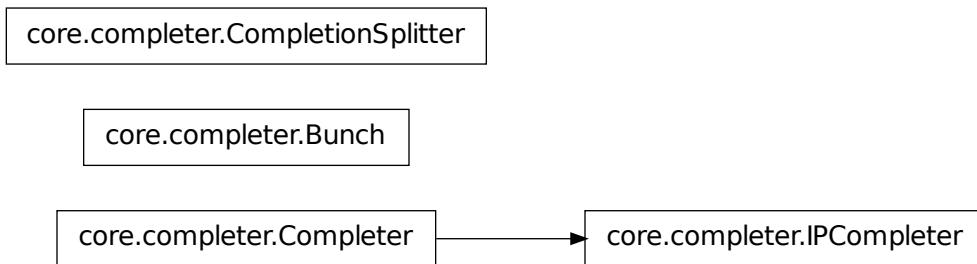
Compute a (probably) unique name for code for caching.

This now expects code to be unicode.

8.9 core.completer

8.9.1 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.

8.9.2 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.

8.9.3 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.

8.10 core.completerlib

8.10.1 Module: core.completerlib

Implementations for various useful completers.

These are all loaded by default by IPython.

8.10.2 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)`

8.11 core.crashhandler

8.11.1 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

8.11.2 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.

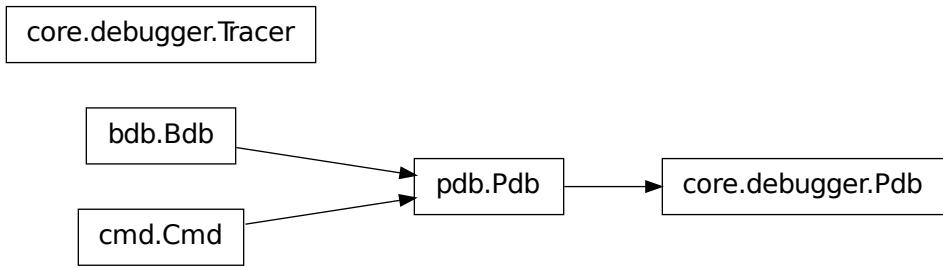
message_template = "Oops, {app_name} crashed. We do our best to make it stable, but...\\n\\nA crash report"

section_sep = '\\n\\n*****

8.12 core.debugger

8.12.1 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>

8.12.2 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).

commands_resuming = ['do_continue', 'do_step', 'do_next', 'do_return', 'do_quit', 'do_jump']

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*)

displayhook (*obj*)

Custom displayhook for the exec in default(), which prevents assignment of the _ variable in the builtins.

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_c1 (*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_whatis (*arg*)
do_where (*arg*)
doc_header = ‘Documented commands (type help <topic>):’
doc_leader = ‘

`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_whatis()
help_where()
identchars = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
```

```
interaction(frame, traceback)
intro = None
lastcmd =
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.
misc_header = 'Miscellaneous help topics:'
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)
nohelp = '*** No help on %s'
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)
prompt = '(Cmd) '
reset()
ruler = '='
run(cmd, globals=None, locals=None)
runcall(func, *args, **kwds)
runctx(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_stepset_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)
undoc_header = 'Undocumented commands:'

use_rawinput = 1

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)
    This function is called if an exception occurs, but only if we are to stop at or just below this
    level.
```

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.

8.12.3 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.

8.13 core.display

8.13.1 Module: core.display

Top-level display functions for displaying object in different formats.

Authors:

- Brian Granger

8.13.2 Functions

`IPython.core.display.display(*objs, **kwargs)`

Display a Python object in all frontends.

By default all representations will be computed and sent to the frontends. Frontends can decide which representation is used and how.

Parameters `objs` : tuple of objects

The Python objects to display.

`include` : list or tuple, optional

A list of format type strings (MIME types) to include in the format data dict.

If this is set *only* the format types included in this list will be computed.

`exclude` : list or tuple, optional

A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

`IPython.core.display.display_html(*objs)`

Display the HTML representation of an object.

Parameters `objs` : tuple of objects

The Python objects to display.

`IPython.core.display.display_javascript(*objs)`

Display the Javascript 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.

8.14 core.display_trap

8.14.1 Module: core.display_trap

Inheritance diagram for `IPython.core.display_trap`:



A context manager for handling `sys.displayhook`.

Authors:

- Robert Kern
- Brian Granger

8.14.2 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)
classmethod class_config_section()
    Get the config class config section
classmethod class_get_help()
    Get the help string for this class in ReST format.
classmethod class_get_trait_help(trait)
    Get the help string for a single trait.
classmethod class_print_help()
    Get the help string for a single trait and print it.
classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

    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.

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.

created = None

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 unintall 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.

8.15 core.displayhook

8.15.1 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

8.15.2 DisplayHook

```
class IPython.core.displayhook.DisplayHook(shell=None, cache_size=1000, colors='NoColor', input_sep='\n', output_sep='\n', output_sep2=';', ps1=None, ps2=None, ps_out=None, pad_left=True, config=None)
Bases: IPython.config.configurable.Configurable
```

The custom IPython displayhook to replace sys.displayhook.

This class does many things, but the basic idea is that it is a callable that gets called anytime user code returns a value.

Currently this class does more than just the displayhook logic and that extra logic should eventually be moved out of here.

```
__init__(shell=None, cache_size=1000, colors='NoColor', input_sep='\n', output_sep='\n', output_sep2=';', ps1=None, ps2=None, ps_out=None, pad_left=True, config=None)
```

check_for_underscore()

Check if the user has set the ‘_’ variable by hand.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

compute_format_data(result)

Compute format data of the object to be displayed.

The format data is a generalization of the `repr()` of an object. In the default implementation the format data is a `dict` of key value pair where the keys are valid MIME types and the values are JSON’able data structure containing the raw data for that MIME type. It is up to frontends to determine pick a MIME to use and display that data in an appropriate manner.

This method only computes the format data for the object and should NOT actually print or write that to a stream.

Parameters `result` : object

The Python object passed to the display hook, whose format will be computed.

Returns `format_data` : dict

A `dict` whose keys are valid MIME types and values are JSON’able raw data for that MIME type. It is recommended that all return values of this should always include the “text/plain” MIME type representation of the object.

config

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

created = None

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 unintall 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.stdout`. 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.stdout`.

8.16 core.displaypub

8.16.1 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

8.16.2 DisplayPublisher

```
class IPython.core.displaypub.DisplayPublisher(**kwargs)
    Bases: IPython.config.configurable.Configurable
```

A traited 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.

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`on_trait_change(handler, name=None, remove=False)`**

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_Traitname_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

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, 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.

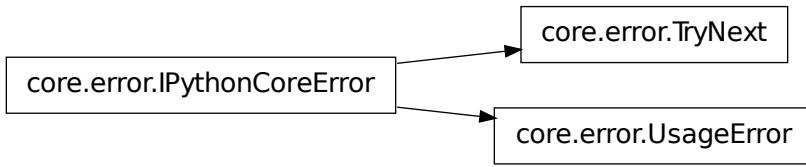
`metadata` : dict

A dictionary for metadata related to the data. This can contain arbitrary key, value pairs that frontends can use to interpret the data.

8.17 core.error

8.17.1 Module: `core.error`

Inheritance diagram for `IPython.core.error`:



Global exception classes for IPython.core.

Authors:

- Brian Granger
- Fernando Perez

Notes

8.17.2 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
```

8.18 core.excolors

8.18.1 Module: core.excolors

Color schemes for exception handling code in IPython.

```
IPython.core.excolors.exception_colors()
Return a color table with fields for exception reporting.
```

The table is an instance of ColorSchemeTable with schemes added for ‘Linux’, ‘LightBG’ and ‘No-Color’ and fields for exception handling filled in.

Examples:

```
>>> ec = exception_colors()
>>> ec.active_scheme_name
''

>>> print ec.active_colors
None
```

Now we activate a color scheme: >>> ec.set_active_scheme('NoColor')>>> ec.active_scheme_name
‘NoColor’ >>> ec.active_colors.keys() [‘em’, ‘filenameEm’, ‘excName’, ‘valEm’, ‘nameEm’, ‘line’,
‘topline’, ‘name’, ‘caret’, ‘val’, ‘vName’, ‘Normal’, ‘filename’, ‘linenoEm’, ‘lineno’, ‘normalEm’]

8.19 core.extensions

8.19.1 Module: core.extensions

Inheritance diagram for IPython.core.extensions:



A class for managing IPython extensions.

Authors:

- Brian Granger

8.19.2 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)

classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.
```

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`

`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 unintall 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)

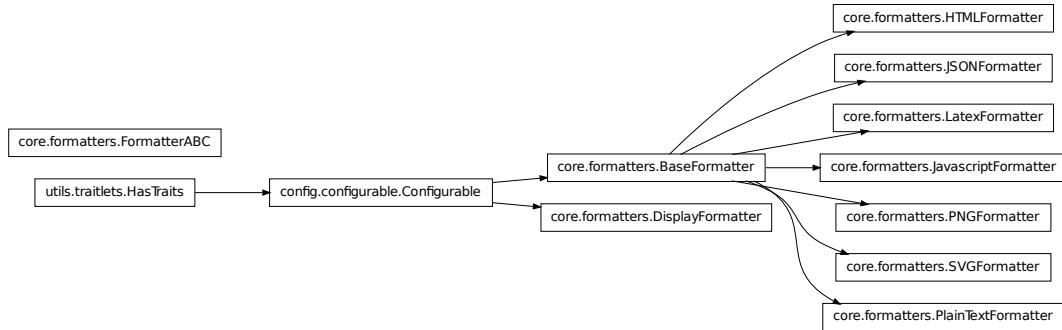
Unload an IPython extension by its module name.

This function looks up the extension's name in sys.modules and simply calls mod.unload_ipython_extension(self).

8.20 core.formatters

8.20.1 Module: core.formatters

Inheritance diagram for IPython.core.formatters:



Display formatters.

Authors:

- Robert Kern
- Brian Granger

8.20.2 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 is 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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

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.

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.

created = None

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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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 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.

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 unintall 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__(self, config=None)
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
enabled = True
format_type = 'text/plain'
```

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 `_repr_html_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be a valid HTML snippet that could be injected into an existing DOM. It should *not* include the '`<html>`' or '`<body>`' tags.

```
__init__(self, config=None)
    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.

```
classmethod class_config_section()
    Get the config class config section
```

```
classmethod class_get_help()
    Get the help string for this class in ReST format.
```

```
classmethod class_get_trait_help(trait)
    Get the help string for a single trait.
```

```
classmethod class_print_help()
    Get the help string for a single trait and print it.
```

classmethod `class_trait_names` (`**metadata`)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (`**metadata`)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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 `_repr_json_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be a valid JSON string.

```
__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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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.

JavascriptFormatter

class IPython.core.formatters.JavascriptFormatter (***kwargs*)
Bases: IPython.core.formatters.BaseFormatter

A Javascript formatter.

To define the callables that compute the Javascript representation of your objects, define a `_repr_javascript_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be valid Javascript code and should *not* be enclosed in '`<script>`' tags.

`__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.

classmethod class_config_section()
Get the config class config section

classmethod class_get_help()
Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
Get the help string for a single trait.

classmethod class_print_help()
Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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 `_repr_latex_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be a valid LaTeX equation, enclosed in either '\$' or '\$\$'.

`__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.

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

`classmethod class_print_help()`

Get the help string for a single trait and print it.

`classmethod class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

`classmethod class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`

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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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 `_repr_png_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be raw PNG data, *not* base64 encoded.

__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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

singleton_printers

An instance of a Python dict.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

type_printers

An instance of a Python dict.

PlainTextFormatter**class IPython.core.formatters.PlainTextFormatter(**kwargs)**

Bases: [IPython.core.formatters.BaseFormatter](#)

The default pretty-printer.

This uses `IPython.external.pretty` to compute the format data of the object. If the object cannot be pretty printed, `repr()` is used. See the documentation of `IPython.external.pretty` for details on how to write pretty printers. Here is a simple example:

```
def dtype_pprinter(obj, p, cycle):
    if cycle:
        return p.text('dtype(...)')
    if hasattr(obj, 'fields'):
        if obj.fields is None:
            p.text(repr(obj))
        else:
            p.begin_group(7, 'dtype([')
            for i, field in enumerate(obj.descr):
                if i > 0:
                    p.text(',')
                    p.breakable()
                p.pretty(field)
            p.end_group(7, '])')
```

__init__(kwargs)**

Create a configurable given a config config.

Parameters `config` : Config

If this is empty, default values are used. If config is a Config instance, it will be used to configure the instance.

Notes

Subclasses of Configurable must call the `__init__()` method of Configurable *before* doing anything else and using `super()`:

```
class MyConfigurable(Configurable):
    def __init__(self, config=None):
        super(MyConfigurable, self).__init__(config)
        # Then any other code you need to finish initialization.
```

This ensures that instances will be configured properly.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

deferred_printers

An instance of a Python dict.

enabled

A boolean (True, False) trait.

float_format

A trait for unicode strings.

float_precision

A casting version of the unicode 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 unicode strings.

max_width

A integer trait.

newline

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 unintall it.

pprint

A boolean (True, False) trait.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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 `_repr_svg_()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be valid SVG enclosed in '`<svg>`' tags, that could be injected into an existing DOM. It should *not* include the '`<html>`' or '`<body>`' tags.

`__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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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 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 unintall it.

print_method

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

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.

8.20.3 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.

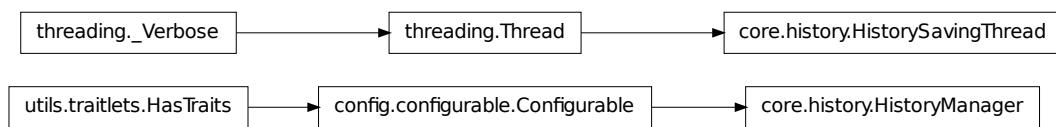
exclude : list or tuple, optional

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.

8.21 core.history

8.21.1 Module: `core.history`

Inheritance diagram for `IPython.core.history`:



History related magics and functionality

8.21.2 Classes

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(*trait*)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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(conn=None)`

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 unintall it.

output_hist

An instance of a Python dict.

output_hist_reprs

An instance of a Python dict.

reset (*new_session=True*)

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

save_flag

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

save_thread

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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.

writetoout_cache (*conn=None*)

Write any entries in the cache to the database.

HistorySavingThread

class IPython.core.history.**HistorySavingThread** (*history_manager*)

Bases: `threading.Thread`

This thread takes care of writing history to the database, so that the UI isn't held up while that happens.

It waits for the HistoryManager's `save_flag` to be set, then writes out the history cache. The main thread is responsible for setting the flag when the cache size reaches a defined threshold.

__init__ (*history_manager*)

`daemon = True`

`getName()`

`ident`

`isAlive()`

`isDaemon()`

`is_alive()`

`join` (*timeout=None*)

`name`

```
run()
setDaemon(daemonic)
setName(name)
```

```
start()
```

```
stop()
```

This can be called from the main thread to safely stop this thread.

Note that it does not attempt to write out remaining history before exiting. That should be done by calling the HistoryManager's end_session method.

```
stop_now = False
```

8.21.3 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. %recall and %rep are equivalent.

- %recall (no arguments):

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

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

%recall 45

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

%recall 1-4

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

%recall foo+bar

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

```
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

8.22 core.hooks

8.22.1 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.

8.22.2 Class

8.22.3 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

8.22.4 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 linenumber.

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, linenumber, 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’,youfunc) 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

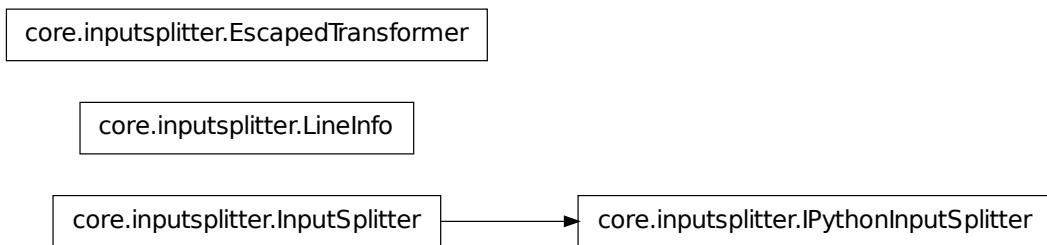
Typically, shutdown hooks should raise TryNext so all shutdown ops are done

`IPython.core.hooks.synchronize_with_editor(self, filename, linenum, column)`

8.23 core.inputsplitter

8.23.1 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

8.23.2 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)
```

```
code = None
```

```
encoding = ''
```

```
indent_spaces = 0
```

```
input_mode = 'line'
```

```
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).

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 = ''
```

```
source_raw = ''
```

```
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 accumulate lines of Python source before execution.

This object is designed to be fed python source line-by-line, using `push()`. 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.

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:\n', 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.

```
code = None
encoding =
indent_spaces = 0
input_mode = 'line'
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).

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 = ''`**`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)

8.23.3 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.has_comment(src)`

Indicate whether an input line has (i.e. ends in, or is) a comment.

This uses tokenize, so it can distinguish comments from # inside strings.

Parameters `src` : string

A single line input string.

Returns Boolean: True if source has a comment. :

`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.

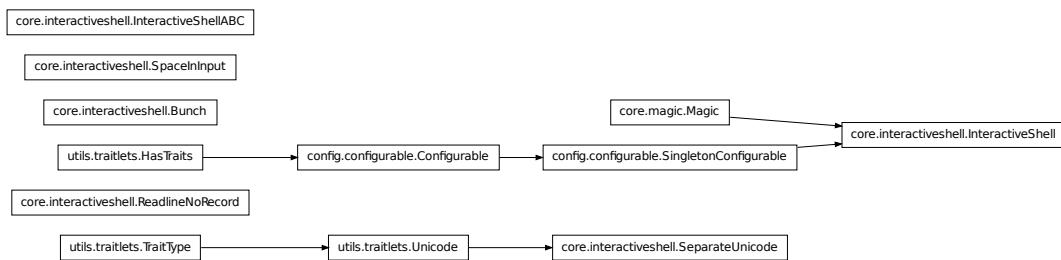
IPython.core.inputsplitter.**transform_help_end**(*line*)
Translate lines with '?/??' at the end

IPython.core.inputsplitter.**transform_ipy_prompt**(*line*)
Handle inputs that start classic IPython prompt syntax.

8.24 core.interactiveshell

8.24.1 Module: core.interactiveshell

Inheritance diagram for IPython.core.interactiveshell:



Main IPython class.

8.24.2 Classes

Bunch

`class IPython.core.interactiveshell.Bunch`

InteractiveShell

`class IPython.core.interactiveshell.InteractiveShell(config=None, ipython_dir=None, profile_dir=None, user_ns=None, user_global_ns=None, custom_exceptions=((), None))`

Bases: `IPython.config.configurable.SingletonConfigurable, IPython.core.magic.Magic`

An enhanced, interactive shell for Python.

`__init__(config=None, ipython_dir=None, profile_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.

auto_status = ['Automagic is OFF, % prefix IS needed for magic functions.', 'Automagic is ON, % prefix N**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

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

`classmethod class_print_help()`

Get the help string for a single trait and print it.

`classmethod class_trait_names(**metadata)`

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

`classmethod class_traits(**metadata)`

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`cleanup()`

`classmethod clear_instance()`

unset _instance for this class and singleton parents.

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.

created = None**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>' % pa-  
rameter_s print 'The self object is:',self
```

```
self.define_magic('foo',foo_Impl)
```

del_var (varname, by_name=False)

Delete a variable from the various namespaces, so that, as far as possible, we’re not keeping any hidden references to it.

Parameters varname : str

The name of the variable to delete.

by_name : bool

If True, delete variables with the given name in each namespace. If False (default), find the variable in the user namespace, and delete references to it.

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, import_all=True*)

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.

exiter

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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 -> standard python form, means including items N...(M-1).

N-M -> 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(*strng*)

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_profile_dir(profile_dir)
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()
Has an instance been created?

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 instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

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*, *next_input=None*)

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_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 argumentsl 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 %book-
mark -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 <book-
mark_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 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_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 changed 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_loadpy` (*arg_s*)

Load a .py python script into the GUI console.

This magic command can either take a local filename or a url:

`%loadpy myscript.py`

`%loadpy http://www.example.com/myscript.py`

`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]: u'%.3f'
```

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

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

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

```
Out[5]: 3

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

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

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

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

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 sensitve 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_psouce (*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_rehashx (*parameter_s*=‘‘)

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

This version explicitly checks that every entry in \$PATH is a file with execute access (os.X_OK), so it is much slower than %rehash.

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

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 0x488b398>)

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 10000000 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_xdel(*parameter_s*=‘‘)

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

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

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.

mkttempfile(*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 unintall 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**profile_dir**

A trait 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_missing_notice(*args, **kwargs)**prompt_in1**

A trait for unicode strings.

prompt_in2

A trait for unicode strings.

prompt_out

A trait for unicode 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 given (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.

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 unicode 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, and attempt to release references to user objects.

If new_session is True, a new history session will be opened.

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.

restore_sys_module_state()

Restore the state of the sys module.

run_ast_nodes(nodelist, cell_name, interactivity='last_expr')

Run a sequence of AST nodes. The execution mode depends on the interactivity parameter.

Parameters `nodelist` : 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', 'last_expr' or 'none', specifying which nodes should be run interactively (displaying output from expressions). 'last_expr' will run the last node interactively only if it is an expression (i.e. expressions in loops or other blocks are not displayed. Other values for this parameter will raise a ValueError.

run_cell(raw_cell, store_history=True)

Run a complete IPython cell.

Parameters `raw_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*)

Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Parameters *code_obj* : code object

A compiled code object, to be executed

post_execute : bool [default: True]

whether to call post_execute hooks after this particular execution.

Returns **False** : successful execution.

True : an error occurred.

runcode (*code_obj*)

Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Parameters *code_obj* : code object

A compiled code object, to be executed

post_execute : bool [default: True]

whether to call post_execute hooks after this particular execution.

Returns **False** : successful execution.

True : an error occurred.

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 Unicode subclass to validate separate_in, separate_out, etc.

This is a Unicode based trait that converts ‘0’->” and ‘n’->”

‘’

separate_out

A Unicode subclass to validate separate_in, separate_out, etc.

This is a Unicode based trait that converts ‘0’->” and ‘n’->”

‘’

separate_out2

A Unicode subclass to validate separate_in, separate_out, etc.

This is a Unicode 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*) → 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

showindentationerror ()

Called by run_cell when there's an IndentationError in code entered at the prompt.

This is overridden in TerminalInteractiveShell to show a message about the %paste magic.

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, piping stdout/err

Parameters cmd : str

Command to execute (can not end in '&', as background processes are not supported. Should not be a command that expects input other than simple text.

system_piped(*cmd*)

Call the given cmd in a subprocess, piping stdout/err

Parameters cmd : str

Command to execute (can not end in '&', as background processes are not supported. Should not be a command that expects input other than simple text.

system_raw(*cmd*)

Call the given cmd in a subprocess using os.system

Parameters cmd : str

Command to execute.

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

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.

SeparateUnicode

```
class IPython.core.interactiveshell.SeparateUnicode (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x489ff50>,  
**metadata)
```

Bases: [IPython.utils.traits.Unicode](#)

A Unicode subclass to validate separate_in, separate_out, etc.

This is a Unicode based trait that converts ‘0’->” and ‘n’->”

:

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x489ff50>,  
**metadata)
```

Create a TraitType.

```
default_value = u''
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a unicode string'
```

```
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.

```
metadata = {}
```

```
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
```

8.24.3 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
```

8.25 core.ipapi

8.25.1 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.
```

8.26 core.logger

8.26.1 Module: core.logger

Inheritance diagram for IPython.core.logger:

```
core.logger.Logger
```

Logger class for IPython's logging facilities.

8.26.2 Logger

```
class IPython.core.logger.Logger(home_dir, logname='Logger.log', loghead='', logmode='over')
```

Bases: object

A Logfile class with different policies for file creation

```
__init__(home_dir, logname='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(logname=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.

8.27 core.macro

8.27.1 Module: core.macro

Inheritance diagram for IPython.core.macro:

```
core.macro.Macro
```

Support for interactive macros in IPython

8.27.2 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
```

8.28 core.magic

8.28.1 Module: core.magic

Inheritance diagram for IPython.core.magic:

core.magic.Magic

core.magic.Bunch

core.magic.MacroToEdit

Magic functions for InteractiveShell.

8.28.2 Classes

Bunch

class IPython.core.magic.Bunch

MacroToEdit

class IPython.core.magic.MacroToEdit

Bases: exceptions.ValueError

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args

message

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

auto_status = ['Automagic is OFF, % prefix IS needed for magic functions.', 'Automagic is ON, % prefix N**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_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 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 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_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 changed 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_loadpy (*arg_s*)

Load a .py python script into the GUI console.

This magic command can either take a local filename or a url:

```
%loadpy myscript.py  
%loadpy http://www.example.com/myscript.py
```

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]: u'% .3f'
```

```
In [3]: pi
```

```
Out[3]: 3.142
```

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

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

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

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

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

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

```
In [7]: pi**10
```

```
Out[7]: 9.364805e+04
```

```
In [8]: %precision
```

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

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

```
Out[9]: 93648.047476082982
```

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 sensitve 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_rehashx (*parameter_s*=‘‘)

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

This version explicitly checks that every entry in \$PATH is a file with execute access (os.X_OK), so it is much slower than %rehash.

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

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 0x488b398>)
```

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 ~”

myfile.s, myfile.l and myfile.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_xdel (*parameter_s*=‘‘)

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

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

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)

8.28.3 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.

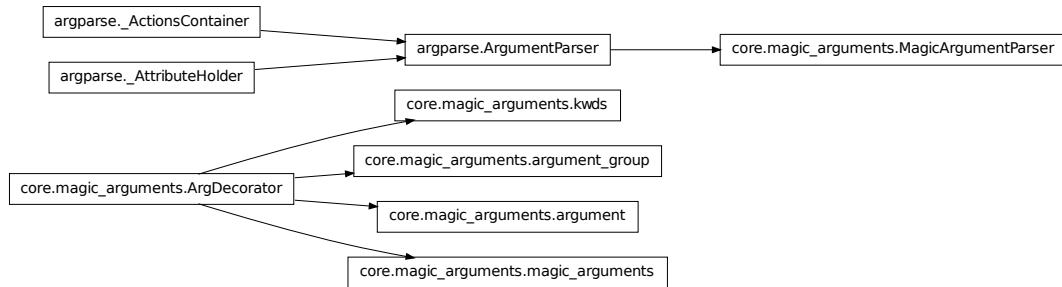
IPython.core.magic.**on_off**(tag)

Return an ON/OFF string for a 1/0 input. Simple utility function.

8.29 core.magic_arguments

8.29.1 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.
```

8.29.2 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,      us-
                                                       age=None,      de-
                                                       scription=None,
                                                       epilog=None,    ver-
                                                       sion=None,      par-
                                                       ents=None,     format-
                                                       ter_class=<class
                                                       'arg-
                                                       parse.HelpFormatter'>,
                                                       prefix_chars='-
                                                       ',           argument_
                                                       default=None,
                                                       con-
                                                       flict_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, par-
          ents=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)
error(message)
    Raise a catchable error instead of exiting.
exit(status=0, message=None)
format_help()
format_usage()
format_version()
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, **kwds)
Bases: IPython.core.magic_arguments.ArgDecorator
```

Store arguments and keywords to pass to add_argument().

Instances also serve to decorate command methods.

```
__init__ (*args, **kwds)
add_to_parser (parser, group)
    Add this object's information to the parser.
```

argument_group

```
class IPython.core.magic_arguments.argument_group (*args, **kwds)
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, **kwds)
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.

8.29.3 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.

8.30 core.oinspect

8.30.1 Module: core.oinspect

Inheritance diagram for `IPython.core.oinspect`:

core.oinspect.Inspect

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.

8.30.2 Class

8.30.3 Inspector

```
class IPython.core.oinspect.Inspector(color_table={'':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x30ee0e0>, 'LightBG':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x30ea170>, 'NoColor':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x30eeb90>, 'Linux':
    <IPython.utils.coloransi.ColorScheme instance
    at 0x30ee0e0>}, code_color_table={'':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x3078758>, 'LightBG':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x3078050>, 'NoColor':
    <IPython.utils.coloransi.ColorScheme
    instance at 0x3078f38>, 'Linux':
    <IPython.utils.coloransi.ColorScheme instance
    at 0x3078758>}, scheme='NoColor',
    str_detail_level=0)

__init__(color_table={'':
    <IPython.utils.coloransi.ColorScheme instance at 0x30ee0e0>,
    'LightBG': <IPython.utils.coloransi.ColorScheme instance at 0x30ea170>,
    'NoColor': <IPython.utils.coloransi.ColorScheme instance at 0x30eeb90>,
    'Linux': <IPython.utils.coloransi.ColorScheme instance at 0x30ee0e0>},
    code_color_table={'':
        <IPython.utils.coloransi.ColorScheme instance at
        0x3078758>, 'LightBG': <IPython.utils.coloransi.ColorScheme instance at
        0x3078050>, 'NoColor': <IPython.utils.coloransi.ColorScheme instance
        at 0x3078f38>, 'Linux': <IPython.utils.coloransi.ColorScheme instance at
        0x3078758>}, 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.

Examples

In [1]: **class NoInit:** ...: pass

In [2]: **class NoDoc:** ...: def __init__(self): ...: pass

In [3]: %pdoc NoDoc No documentation found for NoDoc

In [4]: %pdoc NoInit No documentation found for NoInit

In [5]: obj = NoInit()

In [6]: %pdoc obj No documentation found for obj

In [5]: obj2 = NoDoc()

In [6]: %pdoc obj2 No documentation found for obj2

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.

pinfo_fields1 = [('Type', 'type_name'), ('Base Class', 'base_class'), ('String Form', 'string_form'), ('Name', 'name')]

pinfo_fields_obj = [('Class Docstring', 'class_docstring'), ('Constructor Docstring', 'init_docstring'), ('Class', 'class')]

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*)

8.30.4 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 formattted 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.

8.31 core.page

8.31.1 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. —

8.31.2 Functions

`IPython.core.page.get_pager_cmd (pager_cmd=None)`

Return a pager command.

Makes some attempts at finding an OS-correct one.

`IPython.core.page.get_pager_start (pager, start)`

Return the string for paging files with an offset.

This is the '+N' argument which less and more (under Unix) accept.

`IPython.core.page.page(strng, start=0, screen_lines=0, pager_cmd=None)`

Print a string, piping through a pager after a certain length.

The `screen_lines` parameter specifies the number of *usable* lines of your terminal screen (total lines minus lines you need to reserve to show other information).

If you set `screen_lines` to a number $<=0$, `page()` will try to auto-determine your screen size and will only use up to (`screen_size+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.

8.32 core.payload

8.32.1 Module: core.payload

Inheritance diagram for `IPython.core.payload`:



Payload system for IPython.

Authors:

- Fernando Perez
- Brian Granger

8.32.2 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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**on_trait_change(handler, name=None, remove=False)**

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

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)

8.33 core.payloadpage

8.33.1 Module: core.payloadpage

A payload based version of page.

Authors:

- Brian Granger
- Fernando Perez

8.33.2 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.

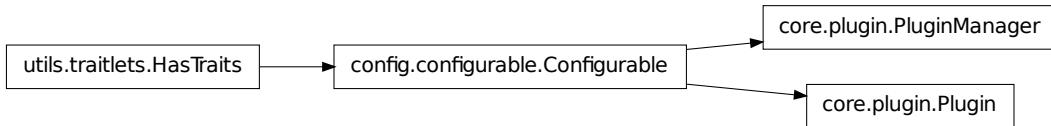
`auto_html` : bool, optional

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.

8.34 core.plugin

8.34.1 Module: core.plugin

Inheritance diagram for `IPython.core.plugin`:



IPython plugins.

Authors:

- Brian Granger

8.34.2 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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(**metadata)`

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(**metadata)`

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If False (the default), then install the handler. If True then unintall it.

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`

A manager for IPython plugins.

__init__ (config=None)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**get_plugin (name, default=None)**

on_trait_change (*handler*, *name=None*, *remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be *handler()*, *handler(name)*, *handler(name, new)* or *handler(name, old, new)*.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

plugins

An instance of a Python dict.

register_plugin (*name*, *plugin*)

trait_metadata (*traitname*, *key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

unregister_plugin (*name*)

8.35 core.prefilter

8.35.1 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

8.35.2 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.

classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

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.

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.

created = None

enabled
    A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)
    Setup a handler to be called when a trait changes.

    This is used to setup dynamic notifications of trait changes.
```

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

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)
```

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

esc_strings

An instance of a Python list.

handle(*line_info*)

Handle alias input lines.

handler_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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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*)

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`enabled`**

A boolean (True, False) trait.

`on_trait_change(handler, name=None, remove=False)`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`enabled`**

A boolean (True, False) trait.

`on_trait_change(handler, name=None, remove=False)`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

`priority`

A integer trait.

`shell`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)
Get metadata values for trait by key.

trait_names (***metadata*)
Get a list of all the names of this classes traits.

traits (***metadata*)
Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

transform (*line, continue_prompt*)

AssignmentChecker

class IPython.core.prefilter.**AssignmentChecker** (*shell=None, filter_manager=None, config=None*)
Bases: IPython.core.prefilter.PrefilterChecker

__init__ (*shell=None, filter_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

classmethod class_config_section()
Get the config class config section

classmethod class_get_help()
Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)
Get the help string for a single trait.

classmethod class_print_help()
Get the help string for a single trait and print it.

classmethod class_trait_names (***metadata*)
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (***metadata*)
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**enabled**

A boolean (True, False) trait.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (**metadata)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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 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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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, filter_manager=None, config=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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

AutocallChecker

```
class IPython.core.prefilter.AutocallChecker(shell=None, filter_manager=None, fig=None)
Bases: IPython.core.prefilter.PrefilterChecker
```

`__init__(shell=None, prefilter_manager=None, config=None)`

`check(line_info)`

Check if the initial word/function is callable and autocall is on.

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

`classmethod class_print_help()`

Get the help string for a single trait and print it.

`classmethod class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

`classmethod class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`

`enabled`

A boolean (True, False) trait.

`on_trait_change(handler, name=None, remove=False)`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod `class_get_trait_help`(*trait*)

Get the help string for a single trait.

classmethod `class_print_help`()

Get the help string for a single trait and print it.

classmethod `class_trait_names`(*metadata*)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`enabled`**

A boolean (True, False) trait.

`on_trait_change`(*handler*, *name=None*, *remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)
```

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**esc_strings**

An instance of a Python list.

handle (line_info)

Handle input lines marked by python-mode.

handler_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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (**metadata)

Get a list of all the names of this classes traits.

traits (**metadata)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

EscCharsChecker

```
class IPython.core.prefilter.EscCharsChecker(shell=None, filter_manager=None, config=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.

classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

    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.
```

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.

created = None**enabled**

A boolean (True, False) trait.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)
```

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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 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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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, config=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

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (`traitname, key`)

Get metadata values for trait by key.

trait_names (**`metadata`)

Get a list of all the names of this classes traits.

traits (**`metadata`)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

IPyPromptTransformer

```
class IPython.core.prefilter.IPyPromptTransformer(shell=None,           pre-
                                                 filter_manager=None,      filter_
                                                 config=None)
```

Bases: `IPython.core.prefilter.PrefilterTransformer`

Handle inputs that start classic IPython prompt syntax.

`__init__(shell=None, prefilter_manager=None, config=None)`

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the [trait_names\(\)](#) method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the [traits\(\)](#) method, but is unbound.

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.

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.

created = None**enabled**

A boolean (True, False) trait.

on_trait_change (handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

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)
```

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

esc_strings

An instance of a Python list.

handle(line_info)

handler_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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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)
```

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**esc_strings**

An instance of a Python list.

handle(line_info)

Execute magic functions.

handler_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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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
```

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

name : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

PrefilterChecker

```
class IPython.core.prefilter.PrefilterChecker(shell=None, filter_manager=None, config=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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`enabled`**

A boolean (True, False) trait.

`on_trait_change(handler, name=None, remove=False)`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_[traitname]_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

PrefilterError**class IPython.core.prefilter.PrefilterError**

Bases: exceptions.Exception

__init__()

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args**message****PrefilterHandler****class IPython.core.prefilter.PrefilterHandler (shell=None,**

pre-

filter_manager=None,

con-

fig=None)

Bases: IPython.config.configurable.Configurable

__init__ (shell=None, prefilter_manager=None, config=None)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`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 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 unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

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.

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

`created = None`

`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 unintall 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*)

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(*trait*)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(*handler*, *name=None*, *remove=False*)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

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)
```

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(handler, name=None, remove=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

transform (*line, continue_prompt*)

PythonOpsChecker

```
class IPython.core.prefilter.PythonOpsChecker (shell=None, filter_manager=None, 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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(*trait*)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

enabled

A boolean (True, False) trait.

on_trait_change(*handler*, *name*=None, *remove*=False)

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention ‘_[traitname]_changed’. Thus, to create static handler for the trait ‘a’, create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters **handler** : callable

A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new) or handler(name, old, new).

name : list, str, None

If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

remove : bool

If False (the default), then install the handler. If True then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (*traitname, key*)

Get metadata values for trait by key.

trait_names (***metadata*)

Get a list of all the names of this classes traits.

traits (***metadata*)

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

ShellEscapeChecker

```
class IPython.core.prefilter.ShellEscapeChecker(shell=None,          pre-
                                                 filter_manager=None,      con-
                                                 config=None)
```

Bases: IPython.core.prefilter.PrefilterChecker

__init__ (*shell=None, prefilter_manager=None, config=None*)

check (*line_info*)

classmethod class_config_section()

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`enabled`**

A boolean (True, False) trait.

`on_trait_change(handler, name=None, remove=False)`

Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '`_Traitname_changed`'. Thus, to create static handler for the trait 'a', create the method `_a_changed(self, name, old, new)` (fewer arguments can be used, see below).

Parameters `handler` : callable

A callable that is called when a trait changes. Its signature can be `handler()`, `handler(name)`, `handler(name, new)` or `handler(name, old, new)`.

`name` : list, str, None

If `None`, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.

`remove` : bool

If `False` (the default), then install the handler. If `True` then unintall it.

prefilter_manager

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

priority

A integer trait.

shell

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

trait_metadata (traitname, key)

Get metadata values for trait by key.

trait_names (metadata)**

Get a list of all the names of this classes traits.

traits (metadata)**

Get a list of all the traits of this class.

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

This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn't exist.

ShellEscapeHandler

```
class IPython.core.prefilter.ShellEscapeHandler(shell=None,          pre-
                                                 filter_manager=None,      con-
                                                 fig=None)                 fig=
```

Bases: IPython.core.prefilter.PrefilterHandler

__init__ (shell=None, prefilter_manager=None, config=None)

classmethod class_config_section ()

Get the config class config section

classmethod class_get_help ()

Get the help string for this class in ReST format.

classmethod class_get_trait_help (trait)

Get the help string for a single trait.

classmethod class_print_help ()

Get the help string for a single trait and print it.

classmethod class_trait_names (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names ()` method, but is unbound.

classmethod `class_traits`(***metadata*)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`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 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 unintall it.

`prefilter_manager`

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

`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.

8.35.3 Function

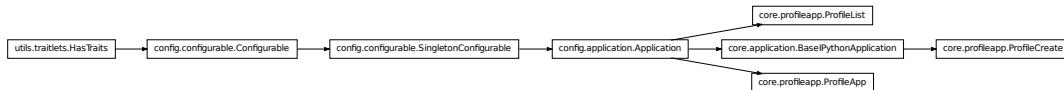
`IPython.core.prefilter.is_shadowed` (*identifier, ip*)

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.

8.36 core.profileapp

8.36.1 Module: core.profileapp

Inheritance diagram for `IPython.core.profileapp`:



An application for managing IPython profiles.

To be invoked as the *ipython profile* subcommand.

Authors:

- Min RK

8.36.2 Classes

ProfileApp

class IPython.core.profileapp.**ProfileApp** (**kwargs)

Bases: IPython.config.application.Application

__init__ (**kwargs)

aliases

An instance of a Python dict.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes

An instance of a Python list.

classmethod clear_instance()

unset _instance for this class and singleton parents.

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.

created = None

description = “Manage IPython profiles\n\nProfile directories contain\nconfiguration, log and security rela

examples = ‘\nipython profile create -h # show the help string for the create subcommand\nipython profile lis

exit (*exit_status*=0)

extra_args

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

initialize(argv=None)

Do the basic steps to configure me.

Override in subclasses.

initialize_subcommand(subc, argv=None)

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

keyvalue_description

A trait for unicode strings.

load_config_file (filename, path=None)

Load a .py based config file by filename and path.

log_level

An enum that whose value must be in a given sequence.

name = u'ipython-profile'**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.

option_description

A trait for unicode strings.

parse_command_line (argv=None)

Parse the command line arguments.

print_alias_help ()

Print the alias part of the help.

print_description ()

Print the application description.

print_examples ()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help ()

Print the flag part of the help.

print_help (classes=False)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()

Print the subcommand part of the help.

print_version()

Print the version string.

start()

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

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.

update_config(config)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

ProfileCreate

class IPython.core.profileapp.ProfileCreate(kwargs)**

Bases: [IPython.core.application.BaseIPythonApplication](#)

__init__(kwargs)**

aliases

An instance of a Python dict.

auto_create

A boolean (True, False) trait.

builtin_profile_dir

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes = [<class 'IPython.core.profiledir.ProfileDir'>]**classmethod clear_instance()**

unset _instance for this class and singleton parents.

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.

config_file_name

A trait for unicode strings.

config_file_paths

An instance of a Python list.

config_file_specified

A boolean (True, False) trait.

config_files

An instance of a Python list.

copy_config_files

A boolean (True, False) trait.

crash_handler_class

A trait whose value must be a subclass of a specified class.

created = None

description = “Create an IPython profile by name\n\nCreate an ipython profile directory by its name or\n\n

examples = ‘\nipython profile create foo # create profile foo w/ default config files\nipython profile create foo -

exit (exit_status=0)**extra_args**

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

init_config_files()**init_crash_handler()**

Create a crash handler, typically setting sys.excepthook to it.

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

init_profile_dir()

initialize the profile dir

initialize(argv=None)**initialize_subcommand(subc, argv=None)**

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using instance, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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_level`

An enum that whose value must be in a given sequence.

`name = u'ipython-profile'`

`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.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parallel

A boolean (True, False) trait.

parse_command_line (argv)**print_alias_help ()**

Print the alias part of the help.

print_description ()

Print the application description.

print_examples ()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help ()

Print the flag part of the help.

print_help (classes=False)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options ()**print_subcommands ()**

Print the subcommand part of the help.

print_version ()

Print the version string.

profile

A trait for unicode strings.

stage_default_config_file ()**start ()**

Start the app mainloop.

Override in subclasses.

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

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.

update_config (config)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

ProfileList**class IPython.core.profileapp.ProfileList (**kwargs)**

Bases: [IPython.config.application.Application](#)

__init__ (kwargs)****aliases**

An instance of a Python dict.

classmethod class_config_section ()

Get the config class config section

classmethod class_get_help ()

Get the help string for this class in ReST format.

classmethod class_get_trait_help (trait)

Get the help string for a single trait.

classmethod class_print_help ()

Get the help string for a single trait and print it.

classmethod class_trait_names (metadata)**

Get a list of all the names of this classes traits.

This method is just like the [trait_names \(\)](#) method, but is unbound.

classmethod class_traits (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes

An instance of a Python list.

classmethod clear_instance()

unset _instance for this class and singleton parents.

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.

created = None

description = “List available IPython profiles\n\nList all available profiles, by profile location, that can\nbe

examples = ‘ipython profile list # list all profiles’

exit (*exit_status*=0)

extra_args

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at `loggin.WARN`, but this can be adjusted by setting the `log_level` attribute.

initialize(argv=None)

Do the basic steps to configure me.

Override in subclasses.

initialize_subcommand(subc, argv=None)

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class `instance`:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`list_profile_dirs()`

`load_config_file(filename, path=None)`

Load a .py based config file by filename and path.

`log_level`

An enum that whose value must be in a given sequence.

`name = u'ipython-profile'`

`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.

option_description

A trait for unicode strings.

parse_command_line (*argv=None*)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help()

Print the flag part of the help.

print_help (*classes=False*)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()

Print the subcommand part of the help.

print_version()

Print the version string.

start()

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

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.

update_config (*config*)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

8.37 core.profiledir

8.37.1 Module: core.profiledir

Inheritance diagram for IPython.core.profiledir:



An object for managing IPython profile directories.

Authors:

- Brian Granger
- Fernando Perez
- Min RK

8.37.2 Classes

ProfileDir

class IPython.core.profiledir.**ProfileDir** (***kwargs*)
Bases: IPython.config.configurable.Configurable

An object to manage the profile directory and its resources.

The profile directory is used by all IPython applications, to manage configuration, logging and security.

This object knows how to find, create and manage these directories. This should be used by any code that wants to handle profiles.

`__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.

`check_dirs()`

`check_log_dir()`

`check_pid_dir()`

`check_security_dir()`

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

`classmethod class_print_help()`

Get the help string for a single trait and print it.

`classmethod class_trait_names(**metadata)`

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

`classmethod class_traits(**metadata)`

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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_config_file (config_file, path=None, overwrite=False)

Copy a default config file into the active profile directory.

Default configuration files are kept in `IPython.config.default`. This function moves these from that location to the working profile directory.

classmethod create_profile_dir (profile_dir, config=None)

Create a new profile directory given a full path.

Parameters profile_dir : str

The full path to the profile directory. If it does exist, it will be used. If not, it will be created.

classmethod create_profile_dir_by_name (path, name=u'default', config=None)

Create a profile dir by profile name and path.

Parameters path : unicode

The path (directory) to put the profile directory in.

name : unicode

The name of the profile. The name of the profile directory will be “profile_<profile>”.

created = None

classmethod find_profile_dir (profile_dir, config=None)

Find/create a profile dir and return its ProfileDir.

This will create the profile directory if it doesn't exist.

Parameters profile_dir : unicode or str

The path of the profile directory. This is expanded using `IPython.utils.genutils.expand_path()`.

classmethod find_profile_dir_by_name (ipython_dir, name=u'default', config=None)

Find an existing profile dir by profile name, return its ProfileDir.

This searches through a sequence of paths for a profile dir. If it is not found, a `ProfileDirError` exception will be raised.

The search path algorithm is: 1. `os.getcwd()` 2. `ipython_dir`

Parameters `ipython_dir` : unicode or str

The IPython directory to use.

`name` : unicode or str

The name of the profile. The name of the profile directory will be “profile_<profile>”.

`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 unintall 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.

ProfileDirError**class IPython.core.profiledir.ProfileDirError**

Bases: exceptions.Exception

__init__()

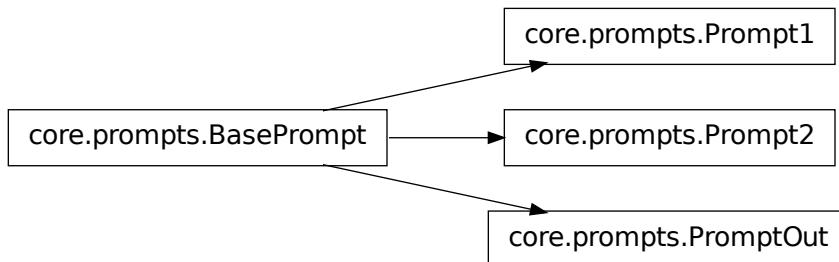
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

args**message**

8.38 core.prompts

8.38.1 Module: core.prompts

Inheritance diagram for IPython.core.prompts:



Classes for handling input/output prompts.

Authors:

- Fernando Perez

- Brian Granger

8.38.2 Classes

BasePrompt

```
class IPython.core.prompts.BasePrompt (cache, sep, prompt, pad_left=False)
Bases: object
```

Interactive prompt similar to Mathematica's.

```
__init__(cache, sep, prompt, pad_left=False)
```

```
cwd_filt(depth)
```

Return the last depth elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

```
cwd_filt2(depth)
```

Return the last depth elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

```
p_template
```

Template for prompt string creation

```
set_p_str()
```

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

```
write(msg)
```

Prompt1

```
class IPython.core.prompts.Prompt1 (cache, sep='n', prompt='In [#: ', pad_left=True)
Bases: IPython.core.prompts.BasePrompt
```

Input interactive prompt similar to Mathematica's.

```
__init__(cache, sep='n', prompt='In [#: ', pad_left=True)
```

```
auto_rewrite()
```

Return a string of the form ‘→’ which lines up with the previous input string. Useful for systems which re-write the user input when handling automatically special syntaxes.

```
cwd_filt(depth)
```

Return the last depth elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

cwd_filt2(*depth*)

Return the last *depth* elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors()**set_p_str()**

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write(*msg*)**Prompt2****class IPython.core.prompts.Prompt2**(*cache, prompt='.\D.: ', pad_left=True*)

Bases: [IPython.core.prompts.BasePrompt](#)

Interactive continuation prompt.

__init__(cache, prompt='.\D.: ', pad_left=True)**cwd_filt**(*depth*)

Return the last *depth* elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

cwd_filt2(*depth*)

Return the last *depth* elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors()**set_p_str()****write**(*msg*)**PromptOut****class IPython.core.prompts.PromptOut**(*cache, sep=', ', prompt='Out[\#]: ', pad_left=True*)

Bases: [IPython.core.prompts.BasePrompt](#)

Output interactive prompt similar to Mathematica’s.

__init__(cache, sep=', ', prompt='Out[\#]: ', pad_left=True)

cwd_filt (depth)

Return the last depth elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

cwd_filt2 (depth)

Return the last depth elements of the current working directory.

\$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

p_template

Template for prompt string creation

set_colors ()**set_p_str ()**

Set the interpolating prompt strings.

This must be called every time the color settings change, because the prompt_specials global may have changed.

write (msg)

8.38.3 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.

8.39 core.shellapp

8.39.1 Module: core.shellapp

Inheritance diagram for IPython.core.shellapp:



A mixin for Application classes that launch InteractiveShell instances, load extensions, etc.

Authors

- Min Ragan-Kelley

8.39.2 InteractiveShellApp

```
class IPython.core.shellapp.InteractiveShellApp(**kwargs)
Bases: IPython.config.configurable.Configurable
```

A Mixin for applications that start InteractiveShell instances.

Provides configurables for loading extensions and executing files as part of configuring a Shell environment.

Provides `init_extensions()` and `init_code()` methods, to be called after `init_shell()`, which must be implemented by subclasses.

`__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.

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(**metadata)`

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`code_to_run`

A trait for unicode strings.

`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.

`created = None`**`exec_files`**

An instance of a Python list.

`exec_lines`

An instance of a Python list.

`extensions`

An instance of a Python list.

`extra_extension`

A trait for unicode strings.

`file_to_run`

A trait for unicode strings.

`init_code()`

run the pre-flight code, specified via `exec_lines`

`init_extensions()`

Load all IPython extensions in `IPythonApp.extensions`.

This uses the `ExtensionManager.load_extensions()` to load all the extensions listed in `self.extensions`.

`init_shell()`**`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.

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.

8.40 core.splitinput

8.40.1 Module: core.splitinput

Simple utility for splitting user input.

Authors:

- Brian Granger
- Fernando Perez

IPython.core.splitinput.**split_user_input** (*line, pattern=None*)

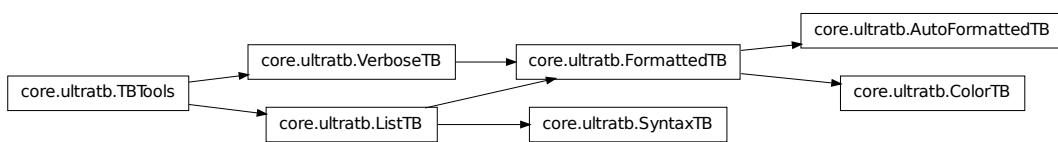
Split user input into pre-char/whitespace, function part and rest.

This is currently handles lines with '=' in them in a very inconsistent manner.

8.41 core.ultratb

8.41.1 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.

8.41.2 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.stdout. This ensures compatibility with most tools, including Windows (where plain stdout doesn’t recognize ANSI escapes).
- Any object with ‘write’ and ‘flush’ attributes.

plain()

set_colors(*args, **kw)

Shorthand access to the color table scheme selector method.

set_mode(*mode=None*)

Switch to the desired mode.

If mode is not specified, cycles through the available modes.

show_exception_only(*etype, evalue*)

Only print the exception type and message, without a traceback.

Parameters *etype* : exception type

value : exception value

stb2text(*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback(*etype=None, value=None, tb=None, tb_offset=None, context=5*)

tb_offset = 0

```
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.stdout`. 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)

tb_offset = 0

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.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.stdout`. 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)
tb_offset = 0
text(etype, value, tb, tb_offset=None, context=5)
    Return formatted traceback.

    Subclasses may override this if they add extra arguments.

verbose()
```

ListTB

```
class IPython.core.ultratb.ListTB(color_scheme='NoColor', call_pdb=False, ostream=None)
Bases: IPython.core.ultratb.TBTools
```

Print traceback information from a traceback list, with optional color.

Calling: requires 3 arguments: (etype, evalue, elist)

as would be obtained by: etype, evalue, tb = sys.exc_info() if tb:

```
    elist = traceback.extract_tb(tb)
```

else: elist = None

It can thus be used by programs which need to process the traceback before printing (such as console replacements based on the code module from the standard library).

Because they are meant to be called without a full traceback (only a list), instances of this class can't call the interactive pdb debugger.

```
__init__(color_scheme='NoColor', call_pdb=False, ostream=None)
```

```
color_toggle()
```

Toggle between the currently active color scheme and NoColor.

```
get_exception_only(etype, value)
```

Only print the exception type and message, without a traceback.

Parameters **etype** : exception type

value : exception value

```
ostream
```

Output stream that exceptions are written to.

Valid values are:

- None: the default, which means that IPython will dynamically resolve

to io.stdout. 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. :

tb_offset = 0

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.stdout. 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. :

tb_offset = 0
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.stdout. This ensures compatibility with most tools, including Windows (where plain stdout doesn't recognize ANSI escapes).

•Any object with ‘write’ and ‘flush’ attributes.

set_colors (**args*, ***kw*)

Shorthand access to the color table scheme selector method.

stb2text (*stb*)

Convert a structured traceback (a list) to a string.

structured_traceback (*etype*, *evalue*, *tb*, *tb_offset=None*, *context=5*, *mode=None*)

Return a list of traceback frames.

Must be implemented by each class.

tb_offset = 0

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 cgitb.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.stdout. This ensures compatibility with most tools, including Windows (where plain stdout doesn’t recognize ANSI escapes).

- Any object with ‘write’ and ‘flush’ attributes.

set_colors(*args, **kw)

Shorthand access to the color table scheme selector method.

stb2text (stb)

Convert a structured traceback (a list) to a string.

structured_traceback (etype, evalue, etb, tb_offset=None, context=5)

Return a nice text document describing the traceback.

tb_offset = 0**text (etype, value, tb, tb_offset=None, context=5)**

Return formatted traceback.

Subclasses may override this if they add extra arguments.

8.41.3 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 ()

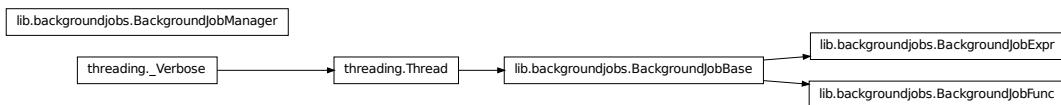
Print a message about internal inspect errors.

These are unfortunately quite common.

8.42 lib.backgroundjobs

8.42.1 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).

8.42.2 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()
```

```
stat_completed = 'Completed'
stat_completed_c = 2
stat_created = 'Created'
stat_created_c = 0
stat_dead = 'Dead (Exception), call jobs.traceback() for details'
stat_dead_c = -1
stat_running = 'Running'
stat_running_c = 1
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()
stat_completed = 'Completed'
stat_completed_c = 2
```

```
stat_created = 'Created'
stat_created_c = 0
stat_dead = 'Dead (Exception), call jobs.traceback() for details'
stat_dead_c = -1
stat_running = 'Running'
stat_running_c = 1
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()
stat_completed = 'Completed'
stat_completed_c = 2
stat_created = 'Created'
stat_created_c = 0
```

```
stat_dead = 'Dead (Exception), call jobs.traceback() for details'  
stat_dead_c = -1  
stat_running = 'Running'  
stat_running_c = 1  
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 function %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 assymmetry 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 (N) → return the result of job N.

status (verbose=0)

Print a status of all jobs currently being managed.

```
traceback(num)
```

8.43 lib.clipboard

8.43.1 Module: lib.clipboard

Utilities for accessing the platform's clipboard.

8.43.2 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.

8.44 lib.deepcopyload

8.44.1 Module: lib.deepcopyload

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.

8.44.2 Functions

```
IPython.lib.deepcopyload.deep_import_hook(name, globals=None, locals=None,  
fromlist=None, level=-1)
```

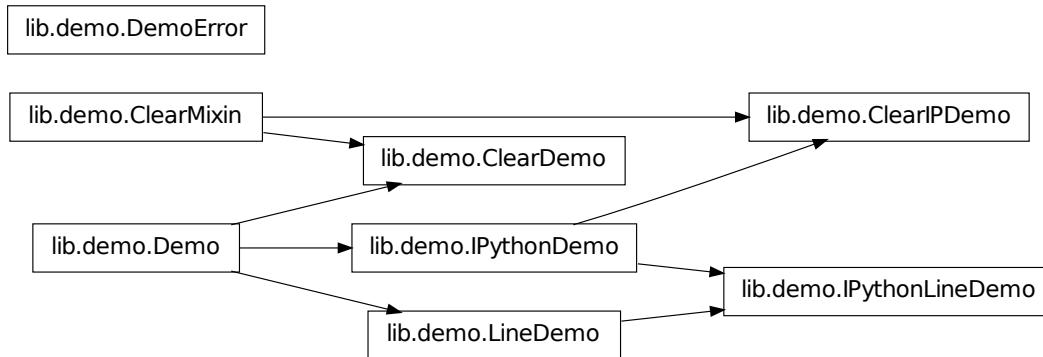
```
IPython.lib.deepreload.deep_reload_hook(module)
IPython.lib.deepreload.determine_parent(globals)
IPython.lib.deepreload.ensure_fromlist(m,fromlist,recursive=0)
IPython.lib.deepreload.find_head_package(parent,name)
IPython.lib.deepreload.import_module(partname,fqname,parent)
IPython.lib.deepreload.load_tail(q,tail)

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.
```

8.45 lib.demo

8.45.1 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 than 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>  
#####
```

8.45.2 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.

`re_auto = <_sre.SRE_Pattern object at 0x4b1e640>`**`re_auto_all = <_sre.SRE_Pattern object at 0x4b3ab50>`****`re_silent = <_sre.SRE_Pattern object at 0x3574e20>`****`re_stop = <_sre.SRE_Pattern object at 0x3574c10>`****`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.

re_auto = <*_sre.SRE_Pattern* object at 0x4b1e640>

re_auto_all = <*_sre.SRE_Pattern* object at 0x4b3ab50>

re_silent = <*_sre.SRE_Pattern* object at 0x3574e20>

re_stop = <*_sre.SRE_Pattern* object at 0x3574c10>

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.

re_auto = <_sre.SRE_Pattern object at 0x4b1e640>**re_auto_all = <_sre.SRE_Pattern object at 0x4b3ab50>****re_silent = <_sre.SRE_Pattern object at 0x3574e20>****re_stop = <_sre.SRE_Pattern object at 0x3574c10>****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.

re_auto = <sre.SRE_Pattern object at 0x4b1e640>**re_auto_all = <sre.SRE_Pattern object at 0x4b3ab50>****re_silent = <sre.SRE_Pattern object at 0x3574e20>****re_stop = <sre.SRE_Pattern object at 0x3574c10>****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.  
  
re_auto = <_sre.SRE_Pattern object at 0x4b1e640>  
re_auto_all = <_sre.SRE_Pattern object at 0x4b3ab50>  
re_silent = <_sre.SRE_Pattern object at 0x3574e20>  
re_stop = <_sre.SRE_Pattern object at 0x3574c10>  
  
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.

re_auto = <_sre.SRE_Pattern object at 0x4b1e640>

re_auto_all = <_sre.SRE_Pattern object at 0x4b3ab50>

re_silent = <_sre.SRE_Pattern object at 0x3574e20>

re_stop = <_sre.SRE_Pattern object at 0x3574c10>

```
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
```

8.45.3 Function

`IPython.lib.demo.re_mark(mark)`

8.46 lib.guisupport

8.46.1 Module: `lib.guisupport`

Support for creating GUI apps and starting event loops.

IPython's GUI integration allows interative 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.

8.46.2 Functions

`IPython.lib.guisupport.get_app_qt4(*args, **kwargs)`

Create a new qt4 app or return an exiting 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.

8.47 lib.inpthook

8.47.1 Module: lib.inpthook

Inheritance diagram for IPython.lib.inpthook:

```
lib.inpthook.InputHookManager
```

Inpthook management for GUI event loop integration.

8.47.2 InputHookManager

```
class IPython.lib.inpthook.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_inpthook(app=None)
```

Set PyOS_InputHook to NULL and return the previous one.

Parameters `app` : optional, ignored

This parameter is allowed only so that clear_inpthook() 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=None)
    Enable event loop integration with PyGTK.
```

Parameters `app` : ignored

Ignored, it's only a placeholder to keep the call signature of all gui activation methods consistent, which simplifies the logic of supporting magics.

Notes

This methods sets the PyOS_InputHook for PyGTK, which allows the PyGTK to integrate with terminal based applications like IPython.

```
enable_qt4(app=None)
    Enable event loop integration with PyQt4.

    Parameters app : Qt Application, optional.
```

Running application to use. If not given, we probe Qt for an existing application object, and create a new one if none is found.

Notes

This methods sets the PyOS_InputHook for PyQt4, which allows the PyQt4 to integrate with terminal based applications like IPython.

If `app` is not given we probe for an existing one, and return it if found. If no existing app is found, we create an `QApplication` as follows:

```
from PyQt4 import QtCore
app = QtGui.QApplication(sys.argv)
```

```
enable_tk(app=None)
    Enable event loop integration with Tk.
```

Parameters `app` : toplevel Tkinter.Tk widget, optional.

Running toplevel widget to use. If not given, we probe Tk for an existing one, and create a new one if none is found.

Notes

If you have already created a Tkinter.Tk object, the only thing done by this method is to register with the `InputHookManager`, since creating that object automatically sets `PyOS_InputHook`.

enable_wx (`app=None`)

Enable event loop integration with wxPython.

Parameters `app` : WX Application, optional.

Running application to use. If not given, we probe WX for an existing application object, and create a new one if none is found.

Notes

This methods sets the `PyOS_InputHook` for wxPython, which allows the wxPython to integrate with terminal based applications like IPython.

If `app` is not given we probe for an existing one, and return it if found. If no existing app is found, we create an `wx.App` as follows:

```
import wx
app = wx.App(redirect=False, clearSigInt=False)
```

get_pyos_inputhook()

Return the current PyOS_InputHook as a ctypes.c_void_p.

get_pyos_inputhook_as_func()

Return the current PyOS_InputHook as a ctypes.PYFUNCYPE.

set_inputhook (`callback`)

Set PyOS_InputHook to callback and return the previous one.

`IPython.lib.inputhook.enable_gui` (`gui=None, app=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, existing application object.

For toolkits that have the concept of a global app, you can supply an existing one. If not given, the toolkit will be probed for one, and if none is found,

a new one will be created. Note that GTK does not have this concept, and passing an app if ‘gui’==”GTK” will raise an error.

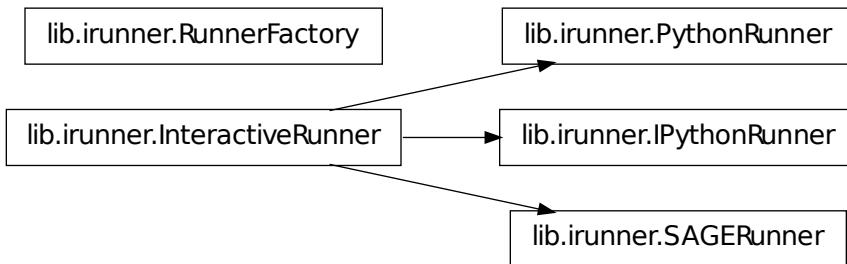
Returns The output of the underlying gui switch routine, typically the actual :

**PyOS_InputHook wrapper object or the GUI toolkit app created, if there was :
one.** :

8.48 lib.irunner

8.48.1 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://mail.scipy.org/pipermail/ipython-user/2006-May/003539.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.

8.48.2 Classes

IPythonRunner

```
class IPython.lib.irunner.IPythonRunner(program='ipython', args=None,
                                         out=<open file '<stdout>', mode 'w'
                                         at 0x2b5e9fa07150>, 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
         0x2b5e9fa07150>, 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 0x2b5e9fa07150>, 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
          0x2b5e9fa07150>, 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
                                         0x2b5e9fa07150>, echo=True)
```

Bases: IPython.lib.irunner.InteractiveRunner

Interactive Python runner.

```
__init__(program='python', args=None, out=<open file '<stdout>', mode 'w' at
         0x2b5e9fa07150>, 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 0x2b5e9fa07150>*)

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 0x2b5e9fa07150>*)

Instantiate a code runner.

SAGERunner

class IPython.lib.irunner.SAGERunner(*program='sage', args=None, out=<open file ‘<stdout>’, mode ‘w’ at 0x2b5e9fa07150>, 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
0x2b5e9fa07150>, 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.

8.48.3 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.

8.49 lib.latextools

8.49.1 Module: lib.latextools

Tools for handling LaTeX.

Authors:

- Brian Granger

8.49.2 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=False)`

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.

`IPython.lib.latextools.math_to_image(s, filename_or_obj, prop=None, dpi=None, format=None)`

Given a math expression, renders it in a closely-clipped bounding box to an image file.

`s` A math expression. The math portion should be enclosed in dollar signs.

`filename_or_obj` A filepath or writable file-like object to write the image data to.

`prop` If provided, a FontProperties() object describing the size and style of the text.

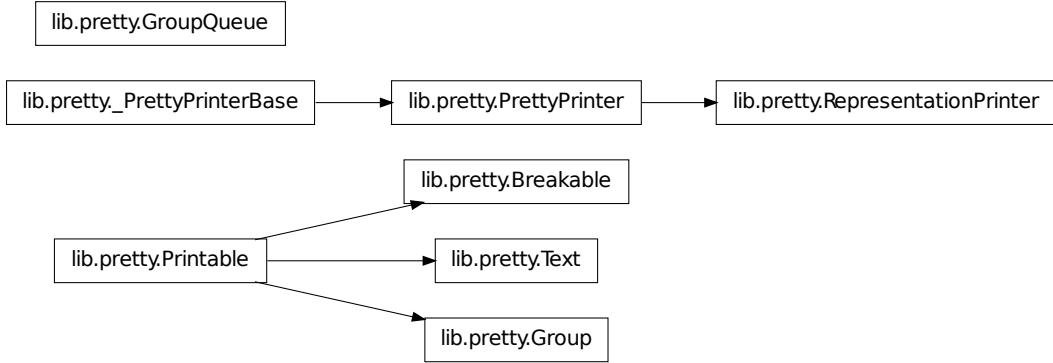
`dpi` Override the output dpi, otherwise use the default associated with the output format.

`format` The output format, eg. ‘svg’, ‘pdf’, ‘ps’ or ‘png’. If not provided, will be deduced from the filename.

8.50 lib.pretty

8.50.1 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 `_repr_pretty_` method to your object and call the methods on the pretty printer passed:

```
class MyObject(object):
    def _repr_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 `_repr_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 _repr_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 bracket of `MyList`.

If you want to support python 2.4 and lower you can use this code:

```
class MyList(list):

    def _repr_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.

8.50.2 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 *_repr_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 have 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*, ***kwds*)

like *begin_group* / *end_group* but for the with statement.

indent(**args*, ***kwds*)

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 reinstantiate 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 have 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*, ***kwds*)

like begin_group / end_group but for the with statement.

indent (**args*, ***kwds*)

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*)

8.50.3 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.
```

8.51 lib.pylabtools

8.51.1 Module: lib.pylabtools

Pylab (matplotlib) support utilities.

Authors

- Fernando Perez.
- Brian Granger

8.51.2 Functions

```
IPython.lib.pylabtools.activate_matplotlib (backend)
```

Activate the given backend and set interactive to True.

```
IPython.lib.pylabtools.figsize (sizex, sizey)
```

Set the default figure size to be [sizex, sizey].

This is just an easy to remember, convenience wrapper that sets:

```
matplotlib.rcParams['figure.figsize'] = [sizex, sizey]
```

```
IPython.lib.pylabtools.find_gui_and_backend (gui=None)
```

Given a gui string return the gui and mpl backend.

Parameters `gui` : str

Can be one of ('tk', 'gtk', 'wx', 'qt', 'qt4', 'inline').

Returns A tuple of (gui, backend) where backend is one of ('TkAgg', 'GTKAgg', :

'WXAgg', 'Qt4Agg', 'module://IPython.zmq.pylab.backend_inline'). :

```
IPython.lib.pylabtools.getfigs (*fig_nums)
```

Get a list of matplotlib figures by figure numbers.

If no arguments are given, all available figures are returned. If the argument list contains references to invalid figures, a warning is printed but the function continues pasting further figures.

Parameters `figs` : tuple

A tuple of ints giving the figure numbers of the figures to return.

```
IPython.lib.pylabtools.import_pylab(user_ns,      backend,      import_all=True,
                                         shell=None)
Import the standard pylab symbols into user_ns.
```

```
IPython.lib.pylabtools.mpl_runner(safe_execfile)
Factory to return a matplotlib-enabled runner for %run.
```

Parameters `safe_execfile` : function

This must be a function with the same interface as the `safe_execfile()` method of IPython.

Returns A function suitable for use as the “runner” argument of the `%run` magic :

function :

```
IPython.lib.pylabtools.print_figure(fig,fmt='png')
Convert a figure to svg or png for inline display.
```

```
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. :

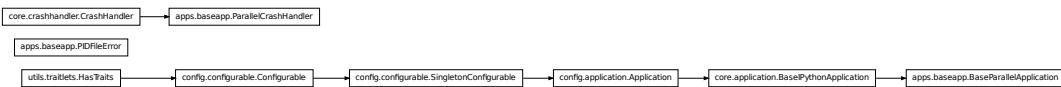
```
IPython.lib.pylabtools.select_figure_format(shell,fmt)
Select figure format for inline backend, either 'png' or 'svg'.
```

Using this method ensures only one figure format is active at a time.

8.52 parallel.apps.baseapp

8.52.1 Module: `parallel.apps.baseapp`

Inheritance diagram for `IPython.parallel.apps.baseapp`:



The Base Application class for IPython.parallel apps

Authors:

- Brian Granger
- Min RK

8.52.2 Classes

`BaseParallelApplication`

`class IPython.parallel.apps.baseapp.BaseParallelApplication(**kwargs)`
Bases: `IPython.core.application.BaseIPythonApplication`

The base Application for IPython.parallel apps

Principle extensions to BaseIPythonApplication:

•`work_dir`

•remote logging via pyzmq

•`IOLoop` instance

`__init__(**kwargs)`

`aliases`

An instance of a Python dict.

`auto_create`

A boolean (True, False) trait.

`builtin_profile_dir`

A trait for unicode strings.

`check_pid(pid)`

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`classes`

An instance of a Python list.

`clean_logs`

A boolean (True, False) trait.

classmethod `clear_instance()`

unset _instance for this class and singleton parents.

`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.

`config_file_name`

A trait for unicode strings.

`config_file_paths`

An instance of a Python list.

`config_file_specified`

A boolean (True, False) trait.

`config_files`

An instance of a Python list.

`copy_config_files`

A boolean (True, False) trait.

`crash_handler_class`

alias of `ParallelCrashHandler`

`created = None`**`description`**

A trait for unicode strings.

examples

A trait for unicode strings.

exit (exit_status=0)**extra_args**

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

get_pid_from_file()

Get the pid from the pid file.

If the pid file doesn't exist a `PIDFileError` is raised.

init_config_files()

[optionally] copy default config files into profile dir.

init_crash_handler()

Create a crash handler, typically setting `sys.excepthook` to it.

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at `logging.WARN`, but this can be adjusted by setting the `log_level` attribute.

init_profile_dir()

initialize the profile dir

initialize(argv=None)

initialize the app

initialize_subcommand(subc, argv=None)

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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_level`

An enum that whose value must be in a given sequence.

`log_to_file`

A boolean (True, False) trait.

`log_url`

A trait for unicode strings.

`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.

`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 unintall it.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line (*argv=None*)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help()

Print the flag part of the help.

print_help (*classes=False*)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()

Print the subcommand part of the help.

print_version()

Print the version string.

profile

A trait for unicode strings.

reinit_logging()

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.

stage_default_config_file()
auto generate default config file, and stage it into the profile.

start()
Start the app mainloop.
Override in subclasses.

subapp
A trait whose value must be an instance of a specified class.
The value can also be an instance of a subclass of the specified class.

subcommand_description
A trait for unicode strings.

subcommands
An instance of a Python dict.

to_work_dir()

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_config(config)
Fire the traits events when the config is updated.

version
A trait for unicode strings.

work_dir
A trait for unicode strings.

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.

PIDFileError

```
class IPython.parallel.apps.baseapp.PIDFileError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

ParallelCrashHandler

```
class IPython.parallel.apps.baseapp.ParallelCrashHandler(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.

    message_template = “Oops, {app_name} crashed. We do our best to make it stable, but...\\n\\nA crash report
    section_sep = ‘\\n\\n*****’
```

8.53 parallel.apps.ipclusterapp

8.53.1 Module: parallel.apps.ipclusterapp

Inheritance diagram for IPython.parallel.apps.ipclusterapp:



The ipcluster application.

Authors:

- Brian Granger
- MinRK

8.53.2 Classes

IPClusterApp

class IPython.parallel.apps.ipclusterapp.**IPClusterApp** (**kwargs)

Bases: IPython.config.application.Application

__init__ (**kwargs)

aliases

An instance of a Python dict.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes

An instance of a Python list.

classmethod clear_instance()

unset _instance for this class and singleton parents.

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.

created = None

description = "Start an IPython cluster for parallel computing.\n\nAn IPython cluster consists of 1 control

examples = '\nipcluster start -n=4 # start a 4 node cluster on localhost\nipcluster start -h # show the help str

exit (*exit_status*=0)

extra_args

An instance of a Python list.

flags

An instance of a Python dict.

generate_config_file()

generate default config file from Configurables

init_logging()

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

initialize(argv=None)

Do the basic steps to configure me.

Override in subclasses.

initialize_subcommand(subc, argv=None)

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
>>> class Bam(Bar):
...     pass
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

keyvalue_description

A trait for unicode strings.

load_config_file (filename, path=None)

Load a .py based config file by filename and path.

log_level

An enum that whose value must be in a given sequence.

name = u'ipcluster'**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.

option_description

A trait for unicode strings.

parse_command_line (argv=None)

Parse the command line arguments.

print_alias_help ()

Print the alias part of the help.

print_description ()

Print the application description.

print_examples ()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help ()

Print the flag part of the help.

print_help (classes=False)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()

Print the subcommand part of the help.

print_version()

Print the version string.

start()

subapp

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands = {‘start’: (‘IPython.parallel.apps.ipclusterapp.IPClusterStart’, “Start an IPython cluster for parallel execution.”)}

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_config(config)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

IPClusterEngines

class IPython.parallel.apps.ipclusterapp.IPClusterEngines(kwargs)**

Bases: [IPython.parallel.apps.baseapp.BaseParallelApplication](#)

__init__(kwargs)**

aliases

An instance of a Python dict.

auto_create

A boolean (True, False) trait.

build_launcher (*clsname*)
import and instantiate a Launcher based on importstring

builtin_profile_dir
A trait for unicode strings.

check_pid (*pid*)

classmethod class_config_section()
Get the config class config section

classmethod class_get_help()
Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)
Get the help string for a single trait.

classmethod class_print_help()
Get the help string for a single trait and print it.

classmethod class_trait_names (***metadata*)
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (***metadata*)
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes
An instance of a Python list.

clean_logs
A boolean (True, False) trait.

classmethod clear_instance()
unset _instance for this class and singleton parents.

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.

config_file_name
A trait for unicode strings.

config_file_paths
An instance of a Python list.

config_file_specified
A boolean (True, False) trait.

config_files
An instance of a Python list.

copy_config_files
A boolean (True, False) trait.

crash_handler_class
alias of ParallelCrashHandler

created=None

daemonize
A boolean (True, False) trait.

default_log_level = 20

description = “Start engines connected to an existing IPython cluster\n\nStart one or more engines to conn

engine_launcher_class
A string holding a valid dotted object name in Python, such as A.b3._c

examples = ‘nipcluster engines –profile=mycluster –n=4 # start 4 engines only’

exit (exit_status=0)

extra_args
An instance of a Python list.

flags
An instance of a Python dict.

generate_config_file()
generate default config file from Configurables

get_pid_from_file()
Get the pid from the pid file.
If the pid file doesn't exist a PIDFileError is raised.

init_config_files()
[optionally] copy default config files into profile dir.

init_crash_handler()
Create a crash handler, typically setting sys.excepthook to it.

init_launchers()

init_logging()
Start logging for this application.
The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the log_level attribute.

init_profile_dir()
initialize the profile dir

```
init_signal()
initialize(argv=None)
initialize_subcommand(subc, argv=None)
    Initialize a subcommand with argv.

classmethod initialized()
    Has an instance been created?

classmethod instance(*args, **kwargs)
    Returns a global instance of this class.

    This method create a new instance if none have previously been created and returns a previously
    created instance is one already exists.

    The arguments and keyword arguments passed to this method are passed on to the __init__()
    method of the class upon instantiation.
```

Examples

Create a singleton class using instance, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable): pass
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable): pass
>>> class Bam(Bar): pass
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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_level`

An enum that whose value must be in a given sequence.

`log_to_file`

A boolean (True, False) trait.

log_url

A trait for unicode strings.

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.

n

A integer trait.

name = u'ipcluster'**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.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line(argv=None)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help()
Print the flag part of the help.

print_help(classes=False)
Print the help for each Configurable class in self.classes.
If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()
Print the subcommand part of the help.

print_version()
Print the version string.

profile
A trait for unicode strings.

reinit_logging()

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.

sigint_handler(signum, frame)

stage_default_config_file()
auto generate default config file, and stage it into the profile.

start()
Start the app for the engines subcommand.

start_engines()

start_logging()

stop_engines()

stop_launchers(r=None)

subapp
A trait whose value must be an instance of a specified class.
The value can also be an instance of a subclass of the specified class.

subcommand_description
A trait for unicode strings.

subcommands
An instance of a Python dict.

to_work_dir()

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_config (*config*)

Fire the traits events when the config is updated.

usage = `None`**version**

A trait for unicode strings.

work_dir

A trait for unicode strings.

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.

IPClusterStart**class** IPython.parallel.apps.ipclusterapp.**IPClusterStart** (***kwargs*)

Bases: IPython.parallel.apps.ipclusterapp.IPClusterEngines

__init__ (***kwargs*)**aliases**

An instance of a Python dict.

auto_create

A boolean (True, False) trait.

build_launcher (*clsnme*)

import and instantiate a Launcher based on importstring

builtin_profile_dir

A trait for unicode strings.

check_pid (*pid*)**classmethod** **class_config_section** ()

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`classes`

An instance of a Python list.

`clean_logs`

A boolean (True, False) trait.

classmethod `clear_instance()`

unset _instance for this class and singleton parents.

`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.

`config_file_name`

A trait for unicode strings.

`config_file_paths`

An instance of a Python list.

`config_file_specified`

A boolean (True, False) trait.

`config_files`

An instance of a Python list.

`controller_launcher_class`

A string holding a valid dotted object name in Python, such as `A.b3._c`

`copy_config_files`

A boolean (True, False) trait.

crash_handler_class
alias of ParallelCrashHandler

created = None

daemonize
A boolean (True, False) trait.

default_log_level = 20

delay
A casting version of the float trait.

description = “Start an IPython cluster for parallel computing\n\nStart an ipython cluster by its profile name”

engine_launcher_class
A string holding a valid dotted object name in Python, such as A.b3._c

examples = ‘\nipython profile create mycluster –parallel # create mycluster profile\nipcluster start –profile=mycluster’

exit (exit_status=0)

extra_args
An instance of a Python list.

flags
An instance of a Python dict.

generate_config_file()
generate default config file from Configurables

get_pid_from_file()
Get the pid from the pid file.
If the pid file doesn’t exist a PIDFileError is raised.

init_config_files()
[optionally] copy default config files into profile dir.

init_crash_handler()
Create a crash handler, typically setting sys.excepthook to it.

init_launchers()

init_logging()
Start logging for this application.
The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the log_level attribute.

init_profile_dir()
initialize the profile dir

init_signal()

initialize(argv=None)

initialize_subcommand(subc, argv=None)
Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance (*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

ipython_dir

A trait for unicode strings.

keyvalue_description

A trait for unicode strings.

load_config_file(suppress_errors=True)

Load the config file.

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_level

An enum that whose value must be in a given sequence.

log_to_file

A boolean (True, False) trait.

log_url

A trait for unicode strings.

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.

n

A integer trait.

name = u'ipcluster'

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.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line(argv=None)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help()

Print the flag part of the help.

print_help(classes=False)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()

```
print_subcommands()
    Print the subcommand part of the help.

print_version()
    Print the version string.

profile
    A trait for unicode strings.

reinit_logging()

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.

reset
    A boolean (True, False) trait.

sigint_handler(signum, frame)

stage_default_config_file()
    auto generate default config file, and stage it into the profile.

start()
    Start the app for the start subcommand.

start_controller()

start_engines()

start_logging()

stop_controller()

stop_engines()

stop_launchers(r=None)

subapp
    A trait whose value must be an instance of a specified class.

    The value can also be an instance of a subclass of the specified class.

subcommand_description
    A trait for unicode strings.

subcommands
    An instance of a Python dict.

to_work_dir()

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_config(config)

Fire the traits events when the config is updated.

usage = None**version**

A trait for unicode strings.

work_dir

A trait for unicode strings.

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.

IPClusterStop**class IPython.parallel.apps.ipclusterapp.IPClusterStop(**kwargs)**

Bases: [IPython.parallel.apps.baseapp.BaseParallelApplication](#)

__init__(kwargs)****aliases**

An instance of a Python dict.

auto_create

A boolean (True, False) trait.

builtin_profile_dir

A trait for unicode strings.

check_pid(pid)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`classes`

An instance of a Python list.

`clean_logs`

A boolean (True, False) trait.

classmethod `clear_instance()`

unset _instance for this class and singleton parents.

`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.

`config_file_name`

A trait for unicode strings.

`config_file_paths`

An instance of a Python list.

`config_file_specified`

A boolean (True, False) trait.

`config_files`

An instance of a Python list.

`copy_config_files`

A boolean (True, False) trait.

`crash_handler_class`

alias of ParallelCrashHandler

`created = None`

`description = "Stop a running IPython cluster\n\nStop a running ipython cluster by its profile name or clu`

`examples = '\nipcluster stop -profile=mycluster # stop a running cluster by profile name\n'`

exit (*exit_status*=0)

extra_args
An instance of a Python list.

flags
An instance of a Python dict.

generate_config_file()
generate default config file from Configurables

get_pid_from_file()
Get the pid from the pid file.
If the pid file doesn't exist a PIDFileError is raised.

init_config_files()
[optionally] copy default config files into profile dir.

init_crash_handler()
Create a crash handler, typically setting sys.excepthook to it.

init_logging()
Start logging for this application.
The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

init_profile_dir()
initialize the profile dir

initialize(argv=None)
initialize the app

initialize_subcommand(subc, argv=None)
Initialize a subcommand with argv.

classmethod initialized()
Has an instance been created?

classmethod instance(*args, **kwargs)
Returns a global instance of this class.
This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.
The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using instance, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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_level`

An enum that whose value must be in a given sequence.

`log_to_file`

A boolean (True, False) trait.

`log_url`

A trait for unicode strings.

`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.

`name = u'ipcluster'`

`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.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line (*argv=None*)

Parse the command line arguments.

print_alias_help ()

Print the alias part of the help.

print_description ()

Print the application description.

print_examples ()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help ()

Print the flag part of the help.

print_help (*classes=False*)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options ()

print_subcommands ()

Print the subcommand part of the help.

print_version ()

Print the version string.

profile

A trait for unicode strings.

reinit_logging ()

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.

signal

A integer trait.

stage_default_config_file()
auto generate default config file, and stage it into the profile.

start()
Start the app for the stop subcommand.

subapp
A trait whose value must be an instance of a specified class.
The value can also be an instance of a subclass of the specified class.

subcommand_description
A trait for unicode strings.

subcommands
An instance of a Python dict.

to_work_dir()

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_config(config)
Fire the traits events when the config is updated.

version
A trait for unicode strings.

work_dir
A trait for unicode strings.

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.

8.53.3 Function

`IPython.parallel.apps.ipclusterapp.launch_new_instance()`
Create and run the IPython cluster.

8.54 parallel.apps.ipcontrollerapp

8.54.1 Module: parallel.apps.ipcontrollerapp

Inheritance diagram for IPython.parallel.apps.ipcontrollerapp:



The IPython controller application.

Authors:

- Brian Granger
- MinRK

8.54.2 IPControllerApp

```

class IPython.parallel.apps.ipcontrollerapp.IPControllerApp (**kwargs)
  Bases: IPython.parallel.apps.baseapp.BaseParallelApplication

  __init__(**kwargs)

  aliases
    An instance of a Python dict.

  auto_create
    A boolean (True, False) trait.

  builtin_profile_dir
    A trait for unicode strings.

  check_pid(pid)

  children
    An instance of a Python list.

  classmethod class_config_section()
    Get the config class config section

  classmethod class_get_help()
    Get the help string for this class in ReST format.

  classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

  classmethod class_print_help()
    Get the help string for a single trait and print it.
  
```

classmethod class_trait_names (**metadata)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (**metadata)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes = [<class 'IPython.core.profiledir.ProfileDir'>, <class 'IPython.zmq.session.Session'>, <class 'IPytho

clean_logs

A boolean (True, False) trait.

classmethod clear_instance ()

unset _instance for this class and singleton parents.

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.

config_file_name

A trait for unicode strings.

config_file_paths

An instance of a Python list.

config_file_specified

A boolean (True, False) trait.

config_files

An instance of a Python list.

copy_config_files

A boolean (True, False) trait.

crash_handler_class

alias of ParallelCrashHandler

created = None

description = 'Start the IPython controller for parallel computing.\n\nThe IPython controller provides a g

do_import_statements ()

examples = 'nipycontroller -ip=192.168.0.1 -port=1000 # listen on ip, port for engines\nnipycontroller -scheme-

exit (exit_status=0)

extra_args

An instance of a Python list.

flags

An instance of a Python dict.

forward_logging()**generate_config_file()**

generate default config file from Configurables

get_pid_from_file()

Get the pid from the pid file.

If the pid file doesn't exist a `PIDFileError` is raised.

import_statements

An instance of a Python list.

init_config_files()

[optionally] copy default config files into profile dir.

init_crash_handler()

Create a crash handler, typically setting `sys.excepthook` to it.

init_hub()**init_logging()**

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at `logging.WARN`, but this can be adjusted by setting the `log_level` attribute.

init_profile_dir()

initialize the profile dir

init_schedulers()**initialize(argv=None)****initialize_subcommand(subc, argv=None)**

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using instance, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

`ipython_dir`

A trait for unicode strings.

`keyvalue_description`

A trait for unicode strings.

`load_config_file(suppress_errors=True)`

Load the config file.

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.

`load_config_from_json()`

load config from existing json connector files.

`location`

A trait for unicode strings.

`log_level`

An enum that whose value must be in a given sequence.

`log_to_file`

A boolean (True, False) trait.

`log_url`

A trait for unicode strings.

`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.

`mq_class`

A trait for unicode strings.

`name = u'ipcontroller'`

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.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line(*argv=None*)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application’s usage.

print_flag_help()

Print the flag part of the help.

print_help(*classes=False*)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()**print_subcommands()**

Print the subcommand part of the help.

print_version()

Print the version string.

profile

A trait for unicode strings.

reinit_logging()**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.

reuse_files

A boolean (True, False) trait.

save_connection_dict(*fname, cdict*)

save a connection dict to json file.

save_urls()

save the registration urls to files.

secure

A boolean (True, False) trait.

ssh_server

A trait for unicode strings.

stage_default_config_file()

auto generate default config file, and stage it into the profile.

start()**subapp**

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

An instance of a Python dict.

to_work_dir()**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_config(config)`

Fire the traits events when the config is updated.

`use_threads`

A boolean (True, False) trait.

`version`

A trait for unicode strings.

`work_dir`

A trait for unicode strings.

`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.

`IPython.parallel.apps.ipcontrollerapp.launch_new_instance()`

Create and run the IPython controller

8.55 parallel.apps.ipengineapp

8.55.1 Module: parallel.apps.ipengineapp

Inheritance diagram for `IPython.parallel.apps.ipengineapp`:



The IPython engine application

Authors:

- Brian Granger
- MinRK

8.55.2 Classes

IPEngineApp

```
class IPython.parallel.apps.ipengineapp.IPEngineApp(**kwargs)
    Bases: IPython.parallel.apps.baseapp.BaseParallelApplication

    __init__(**kwargs)

    aliases
        An instance of a Python dict.

    auto_create
        A boolean (True, False) trait.

    builtin_profile_dir
        A trait for unicode strings.

    check_pid(pid)

    classmethod class_config_section()
        Get the config class config section

    classmethod class_get_help()
        Get the help string for this class in ReST format.

    classmethod class_get_trait_help(trait)
        Get the help string for a single trait.

    classmethod class_print_help()
        Get the help string for a single trait and print it.

    classmethod class_trait_names(**metadata)
        Get a list of all the names of this classes traits.

        This method is just like the trait_names() method, but is unbound.

    classmethod class_traits(**metadata)
        Get a list of all the traits of this class.

        This method is just like the traits() method, but is unbound.

    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.

    classes
        An instance of a Python list.

    clean_logs
        A boolean (True, False) trait.
```

```
classmethod clear_instance()  
    unset _instance for this class and singleton parents.  
  
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.  
  
config_file_name  
    A trait for unicode strings.  
  
config_file_paths  
    An instance of a Python list.  
  
config_file_specified  
    A boolean (True, False) trait.  
  
config_files  
    An instance of a Python list.  
  
copy_config_files  
    A boolean (True, False) trait.  
  
crash_handler_class  
    alias of ParallelCrashHandler  
  
created = None  
  
description  
    A trait for unicode strings.  
  
examples = '\nipengine -ip=192.168.0.1 -port=1000 # connect to hub at ip and port\nipengine -log-to-file -lo  
  
exit (exit_status=0)  
  
extra_args  
    An instance of a Python list.  
  
find_url_file()  
    Set the url file.  
  
    Here we don't try to actually see if it exists for is valid as that is hadled by the connection logic.  
  
flags  
    An instance of a Python dict.  
  
forward_logging()  
  
generate_config_file()  
    generate default config file from Configurables  
  
get_pid_from_file()  
    Get the pid from the pid file.  
  
    If the pid file doesn't exist a PIDFileError is raised.  
  
init_config_files()  
    [optionally] copy default config files into profile dir.
```

init_crash_handler()

Create a crash handler, typically setting sys.excepthook to it.

init_engine()**init_logging()**

Start logging for this application.

The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but this can be adjusted by setting the `log_level` attribute.

init_mpi()**init_profile_dir()**

initialize the profile dir

initialize(argv=None)**initialize_subcommand(subc, argv=None)**

Initialize a subcommand with argv.

classmethod initialized()

Has an instance been created?

classmethod instance(*args, **kwargs)

Returns a global instance of this class.

This method creates a new instance if none have previously been created and returns a previously created instance if one already exists.

The arguments and keyword arguments passed to this method are passed on to the `__init__()` method of the class upon instantiation.

Examples

Create a singleton class using `instance`, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class `instance`:

```
>>> class Bar(SingletonConfigurable):
...     pass
...
>>> class Bam(Bar):
...     pass
...
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True
```

ipython_dir

A trait for unicode strings.

keyvalue_description

A trait for unicode strings.

load_config_file (*suppress_errors=True*)

Load the config file.

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_level

An enum that whose value must be in a given sequence.

log_to_file

A boolean (True, False) trait.

log_url

A trait for unicode strings.

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.

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 unintall it.

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line (*argv=None*)

Parse the command line arguments.

print_alias_help()
Print the alias part of the help.

print_description()
Print the application description.

print_examples()
Print usage and examples.
This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help()
Print the flag part of the help.

print_help(classes=False)
Print the help for each Configurable class in self.classes.
If classes=False (the default), only flags and aliases are printed.

print_options()

print_subcommands()
Print the subcommand part of the help.

print_version()
Print the version string.

profile
A trait for unicode strings.

reinit_logging()

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.

stage_default_config_file()
auto generate default config file, and stage it into the profile.

start()

startup_command
A trait for unicode strings.

startup_script
A trait for unicode strings.

subapp
A trait whose value must be an instance of a specified class.
The value can also be an instance of a subclass of the specified class.

subcommand_description
A trait for unicode strings.

subcommands

An instance of a Python dict.

to_work_dir()**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_config(config)

Fire the traits events when the config is updated.

url_file

A trait for unicode strings.

url_file_name

A trait for unicode strings.

version

A trait for unicode strings.

wait_for_url_file

A float trait.

work_dir

A trait for unicode strings.

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.

MPI

class IPython.parallel.apps.ipengineapp.**MPI**(**kwargs)
Bases: IPython.config.configurable.Configurable

Configurable for MPI initialization

__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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

default_inits

An instance of a Python dict.

init_script

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 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.

use

A trait for unicode strings.

SimpleStruct

8.55.3 Function

IPython.parallel.apps.ipengineapp.launch_new_instance()

Create and run the IPython engine

8.56 parallel.apps.iploggerapp

8.56.1 Module: parallel.apps.iploggerapp

Inheritance diagram for IPython.parallel.apps.iploggerapp:



A simple IPython logger application

Authors:

- MinRK

8.56.2 IPLoggerApp

```
class IPython.parallel.apps.iploggerapp.IPLoggerApp(**kwargs)
    Bases: IPython.parallel.apps.baseapp.BaseParallelApplication

    __init__(**kwargs)

    aliases
        An instance of a Python dict.

    auto_create
        A boolean (True, False) trait.

    builtin_profile_dir
        A trait for unicode strings.

    check_pid(pid)

    classmethod class_config_section()
        Get the config class config section

    classmethod class_get_help()
        Get the help string for this class in ReST format.

    classmethod class_get_trait_help(trait)
        Get the help string for a single trait.

    classmethod class_print_help()
        Get the help string for a single trait and print it.

    classmethod class_trait_names(**metadata)
        Get a list of all the names of this classes traits.

        This method is just like the trait_names() method, but is unbound.
```

classmethod class_traits(***metadata*)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

classes = [<class 'IPython.parallel.apps.logwatcher.LogWatcher'>, <class 'IPython.core.profiledir.ProfileDir'

clean_logs

A boolean (True, False) trait.

classmethod clear_instance()

unset _instance for this class and singleton parents.

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.

config_file_name

A trait for unicode strings.

config_file_paths

An instance of a Python list.

config_file_specified

A boolean (True, False) trait.

config_files

An instance of a Python list.

copy_config_files

A boolean (True, False) trait.

crash_handler_class

alias of `ParallelCrashHandler`

created = None

description = 'Start an IPython logger for parallel computing.\n\nIPython controllers and engines (and you

examples

A trait for unicode strings.

exit(*exit_status=0*)**extra_args**

An instance of a Python list.

flags

An instance of a Python dict.

```
generate_config_file()
    generate default config file from Configurables

get_pid_from_file()
    Get the pid from the pid file.

    If the pid file doesn't exist a PIDFileError is raised.

init_config_files()
    [optionally] copy default config files into profile dir.

init_crash_handler()
    Create a crash handler, typically setting sys.excepthook to it.

init_logging()
    Start logging for this application.

    The default is to log to stdout using a StreamHandler. The log level starts at logging.WARN, but
    this can be adjusted by setting the log_level attribute.

init_profile_dir()
    initialize the profile dir

init_watcher()

initialize(argv=None)

initialize_subcommand(subc, argv=None)
    Initialize a subcommand with argv.

classmethod initialized()
    Has an instance been created?

classmethod instance(*args, **kwargs)
    Returns a global instance of this class.

    This method creates a new instance if none have previously been created and returns a previously
    created instance if one already exists.

    The arguments and keyword arguments passed to this method are passed on to the __init__() method
    of the class upon instantiation.
```

Examples

Create a singleton class using instance, and retrieve it:

```
>>> from IPython.config.configurable import SingletonConfigurable
>>> class Foo(SingletonConfigurable):
...     pass
...
>>> foo = Foo.instance()
>>> foo == Foo.instance()
True
```

Create a subclass that is retrieved using the base class instance:

```
>>> class Bar(SingletonConfigurable): pass
>>> class Bam(Bar): pass
>>> bam = Bam.instance()
>>> bam == Bar.instance()
True

ipython_dir
A trait for unicode strings.

keyvalue_description
A trait for unicode strings.

load_config_file(suppress_errors=True)
Load the config file.

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_level
An enum that whose value must be in a given sequence.

log_to_file
A boolean (True, False) trait.

log_url
A trait for unicode strings.

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.

name = u'iplogger'

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.
```

option_description

A trait for unicode strings.

overwrite

A boolean (True, False) trait.

parse_command_line (*argv=None*)

Parse the command line arguments.

print_alias_help()

Print the alias part of the help.

print_description()

Print the application description.

print_examples()

Print usage and examples.

This usage string goes at the end of the command line help string and should contain examples of the application's usage.

print_flag_help()

Print the flag part of the help.

print_help (*classes=False*)

Print the help for each Configurable class in self.classes.

If classes=False (the default), only flags and aliases are printed.

print_options()**print_subcommands()**

Print the subcommand part of the help.

print_version()

Print the version string.

profile

A trait for unicode strings.

reinit_logging()**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.

stage_default_config_file()

auto generate default config file, and stage it into the profile.

start()**subapp**

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

subcommand_description

A trait for unicode strings.

subcommands

An instance of a Python dict.

to_work_dir()**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_config (config)

Fire the traits events when the config is updated.

version

A trait for unicode strings.

work_dir

A trait for unicode strings.

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.

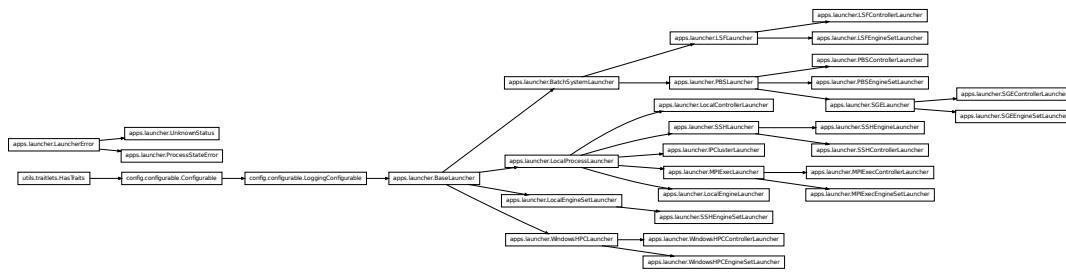
IPython.parallel.apps.iploggerapp.launch_new_instance()

Create and run the IPython LogWatcher

8.57 parallel.apps.launcher

8.57.1 Module: parallel.apps.launcher

Inheritance diagram for IPython.parallel.apps.launcher:



Facilities for launching IPython processes asynchronously.

Authors:

- Brian Granger
- MinRK

8.57.2 Classes

BaseLauncher

```
class IPython.parallel.apps.launcher.BaseLauncher(work_dir=u'',
                                                config=None,
                                                **kwargs)
Bases: IPython.config.configurable.LoggingConfigurable
```

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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

classmethod `class_trait_names` (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`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.

`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 callbacks registered via `on_stop()`.

`on_stop(f)`

Register a callback to be called with this Launcher's `stop_data` when the process actually finishes.

`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.

running

Am I running.

signal (*sig*)

Signal the process.

Parameters `sig` : str or int

‘KILL’, ‘INT’, etc., or any signal number

start()

Start the process.

start_data

stop()

Stop the process and notify observers of stopping.

This method will return None immediately. To observe the actual process stopping, see [on_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 string formatting. 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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod `class_trait_names` (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

default_template

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()

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.

job_array_regexp

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regexp

A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop (*f*)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id (*output*)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regex

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal (*sig*)

Signal the process.

Parameters **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (*n, profile_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’.*, config=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.

classmethod class_config_section ()

Get the config class config section

classmethod `class_get_help()`
Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`
Get the help string for a single trait.

classmethod `class_print_help()`
Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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 unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

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.
```

LSFControllerLauncher

```
class IPython.parallel.apps.launcher.LSFControllerLauncher(work_dir=u'',
                                                          config=None,
                                                          **kwargs)
```

Bases: IPython.parallel.apps.launcher.LSFLauncher

Launch a controller using LSF.

```
__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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regexp

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regexp

A trait for unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(sig)

Signal the process.

Parameters `sig` : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(profile_dir)

Start the controller by profile or profile_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.

LSFEngineSetLauncher

```
class IPython.parallel.apps.launcher.LSFEngineSetLauncher(work_dir=u'.',
                                                       config=None,
                                                       **kwargs)
```

Bases: [IPython.parallel.apps.launcher.LSFLauncher](#)

Launch Engines using LSF

```
__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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regex

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher's stop_data when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regex

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal (sig)

Signal the process.

Parameters **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (n, profile_dir)

Start n engines by profile or profile_dir.

start_data**stop ()****stop_data**

classmethod `class_config_section()`

Get the config class config section

classmethod `class_get_help()`

Get the help string for this class in ReST format.

classmethod `class_get_trait_help(trait)`

Get the help string for a single trait.

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

default_template

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()

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.

job_array_regexp

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(sig)

Signal the process.

Parameters sig : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(n, profile_dir)

Start n copies of the process using LSF batch system. This cant inherit from the base class because bsub expects to be piped a shell script in order to honor the #BSUB directives : bsub < script

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.

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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via [on_stop\(\)](#).

on_stop(f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

poll()

poll_frequency

A integer trait.

running

Am I running.

signal(sig)

start(profile_dir)

Start the controller by profile_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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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.

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 callbacks registered via [on_stop\(\)](#).

on_stop(f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

poll()

poll_frequency

A integer trait.

running

Am I running.

signal(sig)

start(profile_dir)

Start the engine by profile_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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

running

Am I running.

signal (sig)**start (n, profile_dir)**

Start n engines by profile or profile_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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod `class_trait_names` (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`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.

`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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher's stop_data when the process actually finishes.

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.

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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via [on_stop\(\)](#).

on_stop (f)

Register a callback to be called with this Launcher's stop_data when the process actually finishes.

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.

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 (profile_dir)**

Start the controller by profile_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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via [on_stop\(\)](#).

on_stop(f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

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*, *profile_dir*)

Start n engines by profile or profile_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.

MPIExecLauncher

```
class IPython.parallel.apps.launcher.MPIExecLauncher(work_dir=u':', config=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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via [on_stop\(\)](#) .

on_stop(f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regex

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop(f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id(output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regex

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(sig)

Signal the process.

Parameters sig : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(profile_dir)

Start the controller by profile or profile_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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regexp

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regexp

A trait for unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(sig)

Signal the process.

Parameters `sig` : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(n, profile_dir)

Start n engines by profile or profile_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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regex

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher's stop_data when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regex

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal (sig)

Signal the process.

Parameters **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (n, profile_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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()

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.

job_array_regex

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop(f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id (*output*)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal (*sig*)

Signal the process.

Parameters *sig* : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (*profile_dir*)

Start the controller by profile or profile_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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(*trait*)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`default_template`**

A trait for unicode strings.

`delete_command`

An instance of a Python list.

`find_args()`**`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.

`job_array_regex`

A trait for unicode strings.

`job_array_template`

A trait for unicode strings.

`job_id_regex`

A trait for unicode 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.

`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 callbacks registered via `on_stop()`.

on_stop(*f*)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id(*output*)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(*sig*)

Signal the process.

Parameters **sig** : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (*n, profile_dir*)
Start *n* engines by profile or profile_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 trait for unicode strings.

batch_file_name

A trait for unicode strings.

batch_template

A trait for unicode strings.

batch_template_file

A trait for unicode strings.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**default_template**

A trait for unicode strings.

delete_command

An instance of a Python list.

find_args()**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.

job_array_regex

A trait for unicode strings.

job_array_template

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via [on_stop \(\)](#).

on_stop (f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

parse_job_id(*output*)

Take the output of the submit command and return the job id.

queue

A trait for unicode strings.

queue_regexp

A trait for unicode strings.

queue_template

A trait for unicode strings.

running

Am I running.

signal(*sig*)

Signal the process.

Parameters *sig* : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(*n, profile_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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

find_args()

handle_stderr (*fd, events*)

handle_stdout (*fd, events*)

hostname

A trait for unicode strings.

interrupt_then_kill (*delay=2.0*)

Send INT, wait a delay and then send KILL.

location

A trait for unicode 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.

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 callbacks registered via [on_stop\(\)](#).

on_stop (*f*)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

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(profile_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 trait for unicode strings.

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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**find_args()****handle_stderr(fd, events)**

handle_stdout (*fd, events*)

hostname

A trait for unicode strings.

interrupt_then_kill (*delay=2.0*)

Send INT, wait a delay and then send KILL.

location

A trait for unicode 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.

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 callbacks registered via [on_stop\(\)](#).

on_stop (*f*)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

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(profile_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 trait for unicode strings.

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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**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.

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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

running

Am I running.

signal (*sig*)

start (*n, profile_dir*)

Start engines by profile or profile_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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**find_args()****handle_stderr(fd, events)****handle_stdout(fd, events)****hostname**

A trait for unicode strings.

interrupt_then_kill(delay=2.0)

Send INT, wait a delay and then send KILL.

location

A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

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 (profile_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 trait for unicode strings.

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.WindowsHPLauncher

__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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**extra_args**

An instance of a Python list.

find_args()

job_cmd
A trait for unicode strings.

job_file

job_file_name
A trait for unicode strings.

job_id_regex
A trait for unicode 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.

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 callbacks registered via [on_stop\(\)](#).

on_stop(f)
Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

parse_job_id (*output*)

Take the output of the submit command and return the job id.

running

Am I running.

scheduler

A trait for unicode strings.

signal (*sig*)

Signal the process.

Parameters *sig* : str or int

‘KILL’, ‘INT’, etc., or any signal number

start (*profile_dir*)

Start the controller by profile_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.

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**extra_args**

An instance of a Python list.

find_args()

job_cmd

A trait for unicode strings.

job_file

job_file_name

A trait for unicode strings.

job_id_regex

A trait for unicode 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.

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 callbacks registered via [on_stop\(\)](#).

on_stop(f)

Register a callback to be called with this Launcher’s stop_data when the process actually finishes.

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.

parse_job_id(*output*)

Take the output of the submit command and return the job id.

running

Am I running.

scheduler

A trait for unicode strings.

signal(*sig*)

Signal the process.

Parameters *sig* : str or int

‘KILL’, ‘INT’, etc., or any signal number

start(*n, profile_dir*)

Start the controller by profile_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.

classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

    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.

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.

created = None

find_args()

job_cmd
    A trait for unicode strings.

job_file

job_file_name
    A trait for unicode strings.

job_id_regexp
    A trait for unicode 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.

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 callbacks registered via `on_stop()`.

on_stop (f)

Register a callback to be called with this Launcher’s `stop_data` when the process actually finishes.

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.

parse_job_id (output)

Take the output of the submit command and return the job id.

running

Am I running.

scheduler

A trait for unicode strings.

signal(*sig*)

Signal 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*)

8.57.3 Function

`IPython.parallel.apps.launcher.find_job_cmd()`

8.58 parallel.apps.logwatcher

8.58.1 Module: `parallel.apps.logwatcher`

Inheritance diagram for `IPython.parallel.apps.logwatcher`:



A simple logger object that consolidates messages incoming from ipcluster processes.

Authors:

- MinRK

8.58.2 LogWatcher

```
class IPython.parallel.apps.logwatcher.LogWatcher(**kwargs)
Bases: IPython.config.configurable.LoggingConfigurable
```

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)
```

```
classmethod class_config_section()
```

Get the config class config section

```
classmethod class_get_help()
```

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
```

Get the help string for a single trait.

```
classmethod class_print_help()
```

Get the help string for a single trait and print it.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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.

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 unintall 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.

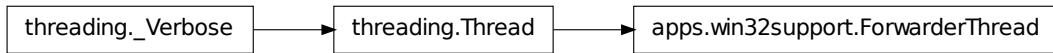
url

A trait for unicode strings.

8.59 parallel.apps.win32support

8.59.1 Module: parallel.apps.win32support

Inheritance diagram for IPython.parallel.apps.win32support:



Utility for forwarding file read events over a zmq socket.

This is necessary because select on Windows only supports sockets, not FDs.

Authors:

- MinRK

8.59.2 ForwarderThread

class IPython.parallel.apps.win32support.**ForwarderThread** (*sock, fd*)

Bases: threading.Thread

__init__ (*sock, fd*)

daemon

getName ()

ident

isAlive ()

isDaemon ()

is_alive ()

```

join(timeout=None)

name

run()
    Loop through lines in self.fd, and send them over self.sock.

setDaemon(daemonic)

setName(name)

start()

IPython.parallel.apps.win32support.forward_read_events(fd,           con-
                                                               text=None)
    Forward read events from an FD over a socket.

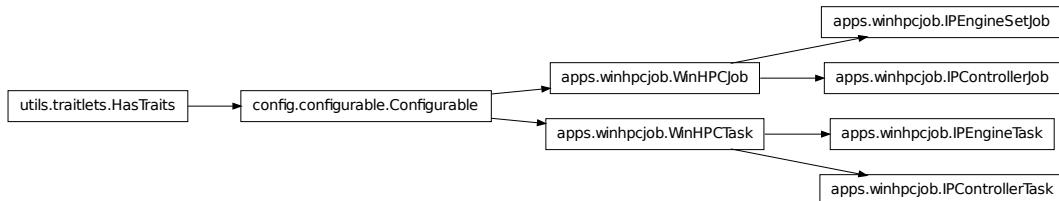
This method wraps a file in a socket pair, so it can be polled for read events by select (specifically
zmq.eventloop.ioloop)

```

8.60 parallel.apps.winhpcjob

8.60.1 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.

Authors:

- Brian Granger
- MinRK

8.60.2 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.

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

`classmethod class_get_trait_help(trait)`

Get the help string for a single trait.

`classmethod class_print_help()`

Get the help string for a single trait and print it.

`classmethod class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

`classmethod class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**is_exclusive**

A boolean (True, False) trait.

job_id

A trait for unicode strings.

job_name

A trait for unicode strings.

job_type

A trait for unicode 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 unintall it.

owner

priority

An enum that whose value must be in a given sequence.

project

A trait for unicode strings.

requested_nodes

A trait for unicode 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 unicode strings.

username

A trait for unicode strings.

version

A trait for unicode strings.

write (*filename*)

Write the XML job description to a file.

xmlns

A trait for unicode strings.

IPControllerTask

class IPython.parallel.apps.winhpcjob.**IPControllerTask** (*config=None*)

Bases: IPython.parallel.apps.winhpcjob.WinHPCTask

__init__ (*config=None*)

as_element ()

classmethod class_config_section ()

Get the config class config section

classmethod class_get_help ()

Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)

Get the help string for a single trait.

classmethod class_print_help ()

Get the help string for a single trait and print it.

classmethod class_trait_names (***metadata*)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (***metadata*)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

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 unintall it.

std_err_file_path

A trait for unicode strings.

std_out_file_path
A trait for unicode strings.

task_id
A trait for unicode strings.

task_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.

unit_type
A trait for unicode strings.

version
A trait for unicode strings.

work_directory
A trait for unicode strings.

IPEngineSetJob

class IPython.parallel.apps.winhpcjob.**IPEngineSetJob** (***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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None

is_exclusive

A boolean (True, False) trait.

job_id

A trait for unicode strings.

job_name

A trait for unicode strings.

job_type

A trait for unicode 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 unintall it.

owner**priority**

An enum that whose value must be in a given sequence.

project

A trait for unicode strings.

requested_nodes

A trait for unicode 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 unicode strings.

username

A trait for unicode strings.

version

A trait for unicode strings.

write (filename)

Write the XML job description to a file.

xmlns

A trait for unicode strings.

IPEngineTask

class IPython.parallel.apps.winhpcjob.**IPEngineTask** (*config=None*)
Bases: IPython.parallel.apps.winhpcjob.WinHPCTask

__init__ (*config=None*)

as_element ()

```
classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the traits() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

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.

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.

created=None
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 unintall it.

std_err_file_path

A trait for unicode strings.

std_out_file_path

A trait for unicode strings.

task_id

A trait for unicode strings.

task_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.

unit_type

A trait for unicode strings.

version

A trait for unicode strings.

work_directory

A trait for unicode strings.

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**is_exclusive**

A boolean (True, False) trait.

job_id

A trait for unicode strings.

job_name

A trait for unicode strings.

job_type

A trait for unicode 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 unintall it.

owner**priority**

An enum that whose value must be in a given sequence.

project

A trait for unicode strings.

requested_nodes

A trait for unicode 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 unicode strings.

username

A trait for unicode strings.

version

A trait for unicode strings.

write (*filename*)

Write the XML job description to a file.

xmlns

A trait for unicode 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()**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

command_line

A trait for unicode strings.

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.

created=None**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 unintall it.

std_err_file_path

A trait for unicode strings.

std_out_file_path

A trait for unicode strings.

task_id

A trait for unicode strings.

task_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.

unit_type

A trait for unicode strings.

version

A trait for unicode strings.

work_directory

A trait for unicode strings.

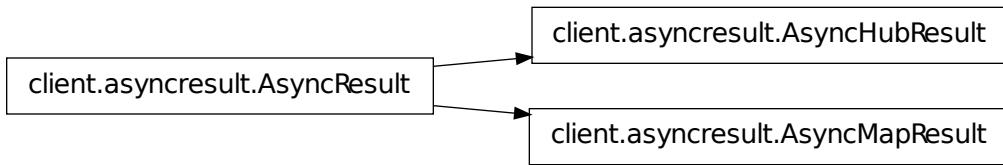
8.60.3 Functions

```
IPython.parallel.apps.winhpcjob.as_str(value)
IPython.parallel.apps.winhpcjob.find_username()
IPython.parallel.apps.winhpcjob.indent(elem, level=0)
```

8.61 parallel.client.asyncresult

8.61.1 Module: parallel.client.asyncresult

Inheritance diagram for IPython.parallel.client.asyncresult:



AsyncResult objects for the client

Authors:

- MinRK

8.61.2 Classes

`AsyncHubResult`

```
class IPython.parallel.client.asyncresult.AsyncHubResult(client, msg_ids,
                                                       fname='unknown',
                                                       targets=None,
                                                       tracker=None)
```

Bases: `IPython.parallel.client.asyncresultAsyncResult`

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.

`msg_ids = None`

`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.

```
msg_ids = None
```

```
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.

msg_ids = None**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.
```

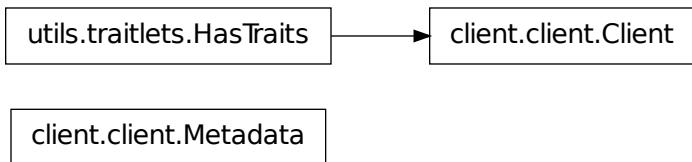
8.61.3 Function

```
IPython.parallel.client.asyncresult.check_ready(f)
    Call spin() to sync state prior to calling the method.
```

8.62 parallel.client.client

8.62.1 Module: `parallel.client.client`

Inheritance diagram for `IPython.parallel.client.client`:



A semi-synchronous Client for the ZMQ cluster

Authors:

- MinRK

8.62.2 Classes

Client

```
class IPython.parallel.client.Client(url_or_file=None, profile=None,
                                      profile_dir=None, ipython_dir=None,
                                      context=None, debug=False,
                                      exec_key=None, sshserver=None,
                                      sshkey=None, password=None,
                                      paramiko=None, timeout=10, **extra_args)
```

Bases: [IPython.utils.traits.HasTraits](#)

A semi-synchronous client to the IPython ZMQ cluster

Parameters **url_or_file** : bytes or unicode; 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. If run from an IPython application, the default profile will be the same as the running application, otherwise it will be 'default'.

context : zmq.Context

Pass an existing zmq.Context instance, otherwise the client will create its own.

debug : bool

flag for lots of message printing for debug purposes

timeout : int/float

time (in seconds) to wait for connection replies from the Hub [Default: 10]

#———— session related args ————— :

config : Config object

If specified, this will be relayed to the Session for configuration

username : str

set username for the session object

packer : str (import_string) or callable

Can be either the simple keyword 'json' or 'pickle', or an import_string to a function to serialize messages. Must support same input as JSON, and output must be bytes. You can pass a callable directly as *pack*

unpacker : str (import_string) or callable

The inverse of packer. Only necessary if packer is specified as *not* one of ‘json’ or ‘pickle’.

#———— 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 signing messages with HMAC digests. :

Messages are still sent as cleartext, so if someone can snoop your :

loopback traffic this will not protect your privacy, but will prevent :

unauthorized execution. :

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=None`, `profile_dir=None`, `ipython_dir=None`, `context=None`, `debug=False`, `exec_key=None`, `sshserver=None`, `sshkey=None`, `password=None`, `paramiko=None`, `timeout=10`, `**extra_args`)

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.

classmethod `class_trait_names` (`**metadata`)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (`**metadata`)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

clear (`targets=None, block=None`)

Clear the namespace in target(s).

close()

db_query (`query, keys=None`)

Query the Hub's TaskRecord database

This will return a list of task record dicts that match `query`

Parameters `query` : mongodb query dict

The search dict. See mongodb query docs for details.

`keys` : list of strs [optional]

The subset of keys to be returned. The default is to fetch everything but buffers. ‘msg_id’ will *always* be included.

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.

hub_history()

Get the Hub's history

Just like the Client, the Hub has a history, which is a list of msg_ids. This will contain the history of all clients, and, depending on configuration, may contain history across multiple cluster sessions.

Any msg_id returned here is a valid argument to `get_result`.

Returns `msg_ids` : list of strs

list of all msg_ids, ordered by task submission time.

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 unintall it.

outstanding

An instance of a Python set.

profile

A trait for unicode strings.

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.

Use `purge_results('all')` to scrub everything from the Hub’s db.

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 int_id, whose entire history is to be purged.

`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

resubmit (`indices_or_msg_ids=None`, `subheader=None`, `block=None`)

Resubmit one or more tasks.

in-flight tasks may not be resubmitted.

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 `AsyncHubResult` :

A subclass of `AsyncResult` that retrieves results from the Hub

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 (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

8.62.3 Function

`IPython.parallel.client.client.spin_first(f)`

Call spin() to sync state prior to calling the method.

8.63 parallel.client.map

8.63.1 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.

Authors:

- Brian Granger
- MinRK

8.63.2 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)
```

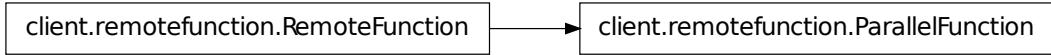
8.63.3 Function

```
IPython.parallel.client.map.mappable (obj)
    return whether an object is mappable or not.
```

8.64 parallel.client.remotefunction

8.64.1 Module: parallel.client.remotefunction

Inheritance diagram for IPython.parallel.client.remotefunction:



Remote Functions and decorators for Views.

Authors:

- Brian Granger
- Min RK

8.64.2 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 than 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, chunkszie=None, **flags)  
block = None  
chunkszie = None  
flags = None  
func = None  
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.  
mapObject = None  
view = None
```

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)
```

block = None

flags = None

func = None

view = None

8.64.3 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
```

8.65 parallel.client.view

8.65.1 Module: parallel.client.view

Inheritance diagram for IPython.parallel.client.view:



Views of remote engines.

Authors:

- Min RK

8.65.2 Classes

DirectView

```
class IPython.parallel.client.view.DirectView(client=None, socket=None, tar-  
                                              gets=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.

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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|AsyncMapResult

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 :
AsyncMapResult An object like AsyncResult, but which reassembles the sequence of results into a single list. AsyncMapResults 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 unintall it.

outstanding

An instance of a Python set.

parallel (*dist*=’b’, *block*=None, ***flags*)

Decorator for making a ParallelFunction

pull (*names*, *targets*=None, *block*=None)

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.

skip_doctest = True

spin()

spin the client, and sync

sync_imports (**args*, ***kwds*)

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*, ***kwds*)

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.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 strs, 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: returnsAsyncResult
    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.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.
    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.
    This method is just like the traits() method, but is unbound.

    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.

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) => listAsyncResult`

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 unintall 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.

retries

A casting version of the int trait.

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.

retries : int

Number of times a task will be retried on failure.

shutdown (*targets=None, restart=False, hub=False, block=None*)

Terminates one or more engine processes, optionally including the hub.

skip_doctest = True

spin()

spin the client, and sync

targets

temp_flags (*args, **kwds)

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.traits.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: returnsAsyncResult

else: returns actual result of f(*args, **kwargs)

apply_async (f, *args, **kwargs)

calls f(*args, **kwargs) on remote engines in a nonblocking manner.

returnsAsyncResult

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.

classmethod class_trait_names (**metadata)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (**metadata)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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 unintall 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.

skip_doctest = True**spin()**

spin the client, and sync

targets**temp_flags**(**args*, ***kwds*)

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

8.65.3 Functions

```
IPython.parallel.client.view.save_ids(f)
Keep our history and outstanding attributes up to date after a method call.

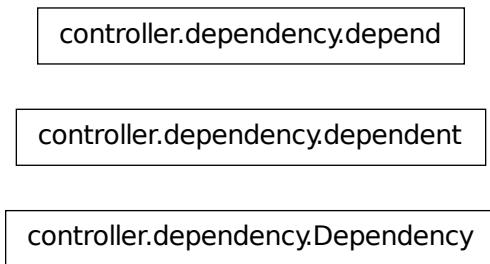
IPython.parallel.client.view.spin_after(f)
call spin after the method.

IPython.parallel.client.view.sync_results(f)
sync relevant results from self.client to our results attribute.
```

8.66 parallel.controller.dependency

8.66.1 Module: parallel.controller.dependency

Inheritance diagram for IPython.parallel.controller.dependency:



Dependency utilities

Authors:

- Min RK

8.66.2 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 orAsyncResult 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.

all = True

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.

failure = True

intersection

Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the 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.

success = True**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)
```

8.66.3 Function

```
IPython.parallel.controller.dependency.require(*mods)
```

Simple decorator for requiring names to be importable.

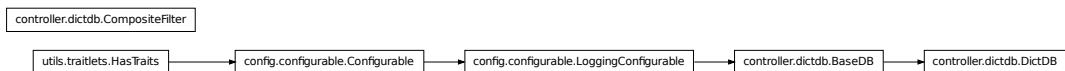
Examples

```
In [1]: @require('numpy') ...: def norm(a): ...: import numpy ...: return numpy.linalg.norm(a,2)
```

8.67 parallel.controller.dictdb

8.67.1 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.

Authors:

- Min RK

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, 'result_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

8.67.2 Classes

BaseDB

```
class IPython.parallel.controller.dictdb.BaseDB(**kwargs)
Bases: IPython.config.configurable.LoggingConfigurable
```

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod `class_trait_names` (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`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.

`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.

`session`

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.

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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**drop_matching_records(check)**

Remove a record from the DB.

drop_record(msg_id)

Remove a record from the DB.

find_records(check, keys=None)

Find records matching a query dict, optionally extracting subset of keys.

Returns dict keyed by msg_id of matching records.

Parameters `check: dict`:

mongodb-style query argument

keys: list of strs [optional]:

if specified, the subset of keys to extract. msg_id will *always* be included.

get_history()
get all msg_ids, ordered by time submitted.

get_record(*msg_id*)
Get a specific Task Record, by msg_id.

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.

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.

session
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.

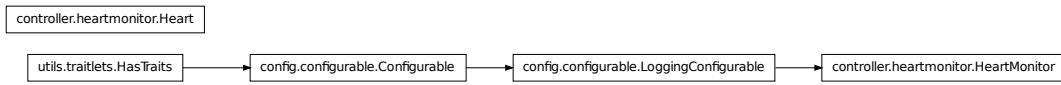
traits(*metadata*)**
Get a list of all the traits of this class.
The TraitTypes returned don’t know anything about the values that the various HasTrait’s instances are holding.
This follows the same algorithm as traits does and does not allow for any simple way of specifying merely that a metadata name exists, but has any value. This is because get_metadata returns None if a metadata key doesn’t exist.

update_record(*msg_id*, *rec*)
Update the data in an existing record.

8.68 parallel.controller.heartmonitor

8.68.1 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.

Authors:

- Min RK

8.68.2 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)
device = None
id = None
start()
```

HeartMonitor

```
class IPython.parallel.controller.heartmonitor.HeartMonitor(**kwargs)
Bases: IPython.config.configurable.LoggingConfigurable
```

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()

classmethod class_config_section()
    Get the config class config section

classmethod class_get_help()
    Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

    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.

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.

created = None

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.

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 unintall 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.

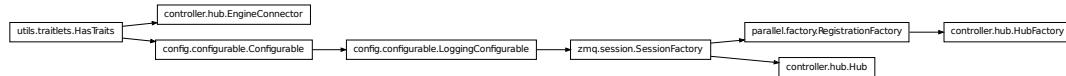
The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

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.

8.69 parallel.controller.hub

8.69.1 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.

Authors:

- Min RK

8.69.2 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*)

classmethod `class_trait_names` (metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits` (metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`control`

A casting version of the string trait.

`heartbeat`

A casting version of the string trait.

`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 casting version of the string trait.

`registration`

A casting version of the string 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.

Hub

class IPython.parallel.controller.hub.**Hub** (***kwargs*)

Bases: IPython.zmq.session.SessionFactory

The IPython Controller Hub with 0MQ connections

Parameters **loop:** zmq IOLoop instance :

session: Session 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*)**classmethod class_config_section()**

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(*trait*)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(***metadata*)

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(***metadata*)

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

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.

created = None

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_query (client_id, msg)

Perform a raw query on the task record database.

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_history (client_id, msg)

Get a list of all msg_ids in our DB records

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 trait for unicode strings.

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 unintall 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

A trait whose value must be an instance of a specified class.

The value can also be an instance of a subclass of the specified class.

resubmit_task (*client_id, msg*)

Resubmit one or more tasks.

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.

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.

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.

unassigned
An instance of a Python set.

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*)

classmethod class_config_section()
Get the config class config section

classmethod class_get_help()
Get the help string for this class in ReST format.

classmethod class_get_trait_help (*trait*)
Get the help string for a single trait.

classmethod class_print_help()
Get the help string for a single trait and print it.

classmethod class_trait_names (***metadata*)
Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits (***metadata*)
Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

client_ip

A trait for unicode strings.

client_transport

A trait for unicode strings.

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()

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

An instance of a Python tuple.

created = None

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 string holding a valid dotted object name in Python, such as A.b3._c

engine_ip

A trait for unicode strings.

engine_transport

A trait for unicode strings.

hb

An instance of a Python tuple.

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.

init_hub()

construct

iopub

An instance of a Python tuple.

ip

A trait for unicode 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 trait for unicode strings.

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 integer trait.

monitor_ip

A trait for unicode strings.

monitor_transport

A trait for unicode strings.

monitor_url

A trait for unicode strings.

mux

An instance of a Python tuple.

notifier_port

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.

report

A integer trait.

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()**task**

An instance of a Python tuple.

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 unicode strings.

url

A trait for unicode strings.

8.69.3 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.

8.70 parallel.controller.scheduler

8.70.1 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.

Authors:

- Min RK

8.70.2 Class

8.70.3 TaskScheduler

```
class IPython.parallel.controller.scheduler.TaskScheduler (**kwargs)
Bases: IPython.zmq.session.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.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod `class_get_trait_help`(*trait*)

Get the help string for a single trait.

classmethod `class_print_help`()

Get the help string for a single trait and print it.

classmethod `class_trait_names`(*metadata*)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits`(*metadata*)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`

`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.

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

hwm

A integer trait.

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 trait for unicode strings.

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.

pending

An instance of a Python dict.

resume_receiving ()

Resume accepting jobs.

retries

An instance of a Python dict.

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.

scheme_name

An enum that whose value must be in a given sequence.

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.

update_graph (*dep_id=None*, *success=True*)
dep_id just finished. Update our dependency graph and submit any jobs that just became runnable.
Called with *dep_id=None* to update entire graph for hwm, but without finishing a task.

8.70.4 Functions

IPython.parallel.controller.scheduler.**launch_scheduler** (*in_addr*,
 out_addr,
 mon_addr,
 not_addr, *con-*
 fig=None, *log-*
 name='root',
 log_url=None,
 loglevel=10,
 identity='task',
 in_thread=False)

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.

8.71 parallel.controller.sqlitedb

8.71.1 Module: parallel.controller.sqlitedb

Inheritance diagram for `IPython.parallel.controller.sqlitedb`:



A TaskRecord backend using sqlite3

Authors:

- Min RK

8.71.2 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.

`classmethod class_config_section()`

Get the config class config section

`classmethod class_get_help()`

Get the help string for this class in ReST format.

```
classmethod class_get_trait_help(trait)
    Get the help string for a single trait.

classmethod class_print_help()
    Get the help string for a single trait and print it.

classmethod class_trait_names(**metadata)
    Get a list of all the names of this classes traits.

    This method is just like the trait_names() method, but is unbound.

classmethod class_traits(**metadata)
    Get a list of all the traits of this class.

    This method is just like the traits() method, but is unbound.

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.

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.

created = None

drop_matching_records(check)
Remove a record from the DB.

drop_record(msg_id)
Remove a record from the DB.

filename
A trait for unicode strings.

find_records(check, keys=None)
Find records matching a query dict, optionally extracting subset of keys.

    Returns list of matching records.

Parameters check: dict :
    mongodb-style query argument

keys: list of strs [optional] :
    if specified, the subset of keys to extract. msg_id will always be included.

get_history()
    get all msg_ids, ordered by time submitted.

get_record(msg_id)
    Get a specific Task Record, by msg_id.
```

location

A trait for unicode 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.

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.

session

A trait for unicode strings.

table

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.

update_record (*msg_id, rec*)

Update the data in an existing record.

8.72 parallel.engine.engine

8.72.1 Module: parallel.engine.engine

Inheritance diagram for IPython.parallel.engine.engine:



A simple engine that talks to a controller over 0MQ. it handles registration, etc. and launches a kernel connected to the Controller's Schedulers.

Authors:

- Min RK

8.72.2 EngineFactory

```
class IPython.parallel.engine.engine.EngineFactory(**kwargs)
Bases: IPython.parallel.factory.RegistrationFactory
```

IPython engine

__init__(kwargs)**

abort()

bident

A casting version of the string trait.

classmethod class_config_section()

Get the config class config section

classmethod class_get_help()

Get the help string for this class in ReST format.

classmethod class_get_trait_help(trait)

Get the help string for a single trait.

classmethod class_print_help()

Get the help string for a single trait and print it.

classmethod class_trait_names(metadata)**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(metadata)**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

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.

created = None**display_hook_factory**

A trait whose value must be a subclass of a specified class.

id

A integer trait.

ident

A trait for unicode strings.

ip

A trait for unicode 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 unicode 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 trait for unicode strings.

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 unintall it.

out_stream_factory

A trait whose value must be a subclass of a specified class.

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 integer trait.

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 unicode strings.

url

A trait for unicode strings.

user_ns

An instance of a Python dict.

8.73 parallel.engine.kernelstarter

8.73.1 Module: parallel.engine.kernelstarter

Inheritance diagram for IPython.parallel.engine.kernelstarter:

```
graph TD; A[engine.kernelstarter.KernelStarter]
```

KernelStarter class that intercepts Control Queue messages, and handles process management.

Authors:

- Min RK

8.73.2 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.

entry point function for launching a kernelstarter in a subprocess

8.74 parallel.engine.streamkernel

8.74.1 Module: parallel.engine.streamkernel

Inheritance diagram for IPython.parallel.engine.streamkernel:



Kernel adapted from kernel.py to use ZMQ Streams

Authors:

- Min RK
- Brian Granger
- Fernando Perez
- Evan Patterson

8.74.2 Kernel

```
class IPython.parallel.engine.streamkernel.Kernel(**kwargs)
    Bases: IPython.zmq.session.SessionFactory

    __init__(**kwargs)

    abort_queue(stream)

    abort_queues()

    abort_request(stream, ident, parent)
        abort a specific msg by id

    aborted
        An instance of a Python set.

    apply_request(stream, ident, parent)

    bident
        A casting version of the string trait.

    check_aborted(msg_id)

    check_dependencies(dependencies)

    classmethod class_config_section()
        Get the config class config section

    classmethod class_get_help()
        Get the help string for this class in ReST format.

    classmethod class_get_trait_help(trait)
        Get the help string for a single trait.
```

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

created = None

dispatch_control (msg)

dispatch_queue (stream, msg)

exec_lines

An instance of a Python list.

execute_request (stream, ident, parent)

ident

A trait for unicode strings.

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 trait for unicode strings.

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 unintall it.

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 ourself. 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.

user_ns

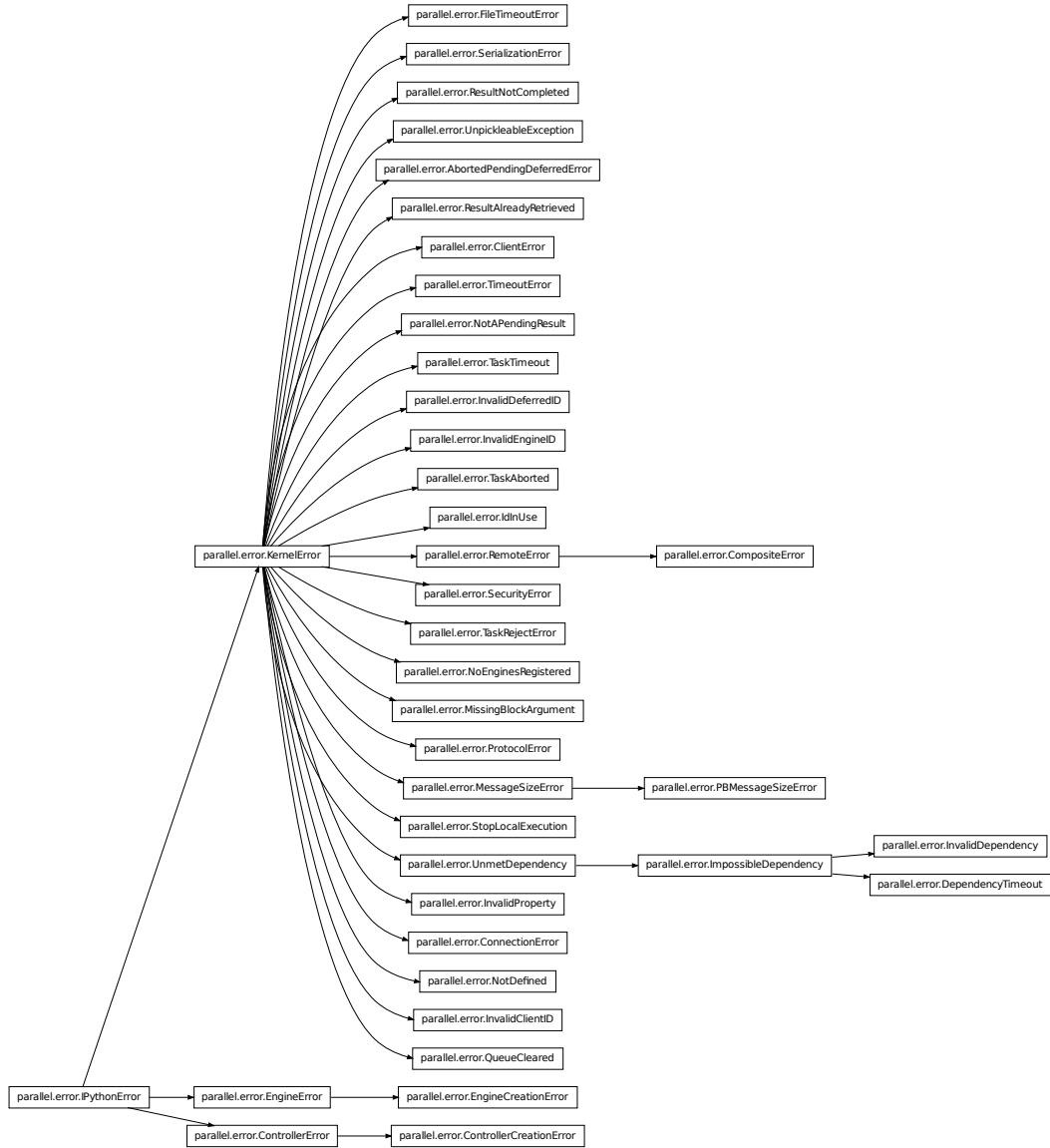
An instance of a Python dict.

`IPython.parallel.engine.streamkernel.printer(*args)`

8.75 parallel.error

8.75.1 Module: parallel.error

Inheritance diagram for `IPython.parallel.error`:



Classes and functions for kernel related errors and exceptions.

Authors:

- Brian Granger
- Min RK

8.75.2 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
    ename = None
    engine_info = None
    evalue = None
    message
    print_tracebacks(excid=None)
    raise_exception(excid=0)
    traceback = None
```

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, evalue, traceback, engine_info=None)
Bases: IPython.parallel.error.KernelError

Error raised elsewhere

__init__(ename, evalue, traceback, engine_info=None)

args
ename = None
engine_info = None
evalue = None
message
traceback = None
```

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
```

8.75.3 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={})
```

8.76 parallel.factory

8.76.1 Module: parallel.factory

Inheritance diagram for IPython.parallel.factory:



Base config factories.

Authors:

- Min RK

8.76.2 RegistrationFactory

```
class IPython.parallel.factory.RegistrationFactory(**kwargs)
Bases: IPython.zmq.session.SessionFactory
```

The Base Configurable for objects that involve registration.

```
__init__(**kwargs)
classmethod class_config_section()
    Get the config class config section
classmethod class_get_help()
    Get the help string for this class in ReST format.
classmethod class_get_trait_help(trait)
    Get the help string for a single trait.
```

classmethod `class_print_help()`

Get the help string for a single trait and print it.

classmethod `class_trait_names(metadata)`**

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod `class_traits(metadata)`**

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

`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.

`created = None`**`ip`**

A trait for unicode 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 trait for unicode strings.

`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 unintall it.

report

A integer trait.

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 unicode strings.

url

A trait for unicode strings.

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

Authors:

- Min RK

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(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(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

itervalues

D.itervalues() -> 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.asbytes(s)`

ensure that an object is ascii bytes

`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.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 TestCase. 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>

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.

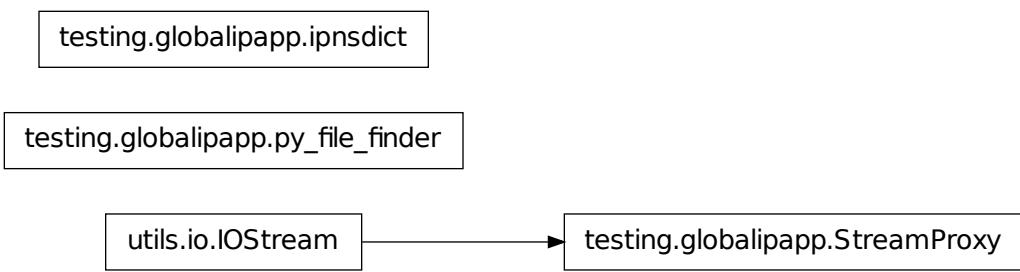
Notes

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:



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

`StreamProxy`

class IPython.testing.globalipapp.**StreamProxy** (*name*)

Bases: IPython.utils.io.IOStream

Proxy for sys.stdout/err. This will request the stream *at call time* allowing for nose's Capture plugin's redirection of sys.stdout/err.

Parameters `name` : str

The name of the stream. This will be requested anew at every call

`__init__(name)`

`close()`

`closed`

`flush()`

`stream`

`write(data)`

`writelines(lines)`

`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(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.ptest

8.81.1 Module: testing.ptest

Inheritance diagram for IPython.testing.ptest:

```
testing.ptest.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.ptest.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 = None  
    list, arguments of system call to be made to call test runner
```

```
params = None  
    list, parameters for test runner
```

```
pids = None  
    list, process ids of subprocesses we start (for cleanup)
```

```
run()  
    Run the stored commands
```

```
runner = None  
    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_ipitest()`

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_ipitestall()`

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 subprocess 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`:

testing.ipunittest.IPython2PythonConverter

testing.ipunittest.Doc2UnitTester

Experimental code for cleaner support of IPython syntax with unittest.

In IPython up until 0.10, we've used very hacked up nose machinery for running tests with IPython special syntax, and this has proved to be extremely slow. This module provides decorators to try a different approach, stemming from a conversation Brian and I (FP) had about this problem Sept/09.

The goal is to be able to easily write simple functions that can be seen by unittest as tests, and ultimately for these to support doctests with full IPython syntax. Nose already offers this based on naming conventions and our hackish plugins, but we are seeking to move away from nose dependencies if possible.

This module follows a different approach, based on decorators.

- A decorator called `@ipdoctest` can mark any function as having a docstring that should be viewed as a doctest, but after syntax conversion.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.82.2 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.

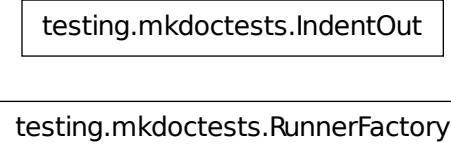
`IPython.testing.ipunittest.ipdocstring(func)`

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  
0x2b5e9fa07150>, 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 0x2b5e9fa07150>, 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 0x2b5e9fa07150>)
```

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 0x2b5e9fa07150>)
```

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 they raise an error, it will be
detected as such, but their output is completely ignored.

>>> 1+3
junk goes here...
>>> 1+3
klasdfj;
```

```
>>> 1+2
again, anything goes
blah...
```

IPython.testing.plugin.dtexample.**ranfunc()**

A function with some random output.

Normal examples are verified as usual: >>> 1+3 4

But if you put '# random' in the output, it is ignored: >>> 1+3 junk goes here... # random

```
>>> 1+2
again, anything goes #random
if multiline, the random mark is only needed once.
```

```
>>> 1+2
You can also put the random marker at the end:
# random
```

```
>>> 1+2
# random
.. or at the beginning.
```

More correct input is properly verified: >>> ranfunc() 'ranfunc'

8.86 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.skipdoctest

8.90.1 Module: `testing.skipdoctest`

This decorator marks that a doctest should be skipped.

The IPython.testing.decorators module triggers various extra imports, including numpy and sympy if they're present. Since this decorator is used in core parts of IPython, it's in a separate module so that running IPython doesn't trigger those imports.

```
IPython.testing.skipdoctest.skip_doctest(f)  
    Decorator - mark a function or method for skipping its doctest.
```

This decorator allows you to mark a function whose docstring you wish to omit from testing, while preserving the docstring for introspection, help, etc.

8.91 testing.tools

8.91.1 Module: `testing.tools`

Inheritance diagram for IPython.testing.tools:

```
graph TD; A[testing.tools.TempFileMixin]
```

Generic testing tools that do NOT depend on Twisted.

In particular, this module exposes a set of top-level assert* functions that can be used in place of nose.tools.assert* in method generators (the ones in nose can not, at least as of nose 0.10.4).

Note: our testing package contains testing.util, which does depend on Twisted and provides utilities for tests that manage Deferreds. All testing support tools that only depend on nose, IPython or the standard library should go here instead.

Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.91.2 Class

8.91.3 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()
```

8.91.4 Functions

```
IPython.testing.tools.check_pairs(func, pairs)
```

Utility function for the common case of checking a function with a sequence of input/output pairs.

Parameters `func` : callable

The function to be tested. Should accept a single argument.

`pairs` : iterable

A list of (input, expected_output) tuples.

Returns `None`. Raises an `AssertionError` if any output does not match the expected

:

`value`. :

```
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.mute_warn(*args, **kwd)`

`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.` :

8.92 utils.PyColorize

8.92.1 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).

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).

8.92.2 Parser

```
class IPython.utils.PyColorize.Parser(color_table=None, out=<open file '<stdout>', mode 'w' at 0x2b5e9fa07150>)
```

Format colored Python source.

```
__init__(color_table=None, out=<open file '<stdout>', mode 'w' at 0x2b5e9fa07150>)
```

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:].

8.93 utils.attic

8.93.1 Module: `utils.attic`

Inheritance diagram for IPython.utils.attic:

utils.attic.EvalDict

utils.attic.NotGiven

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.

8.93.2 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`

8.93.3 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, *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 0x3c44290>)`

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>')`

8.94 utils.autoattr

8.94.1 Module: `utils.autoattr`

Inheritance diagram for `IPython.utils.autoattr`:

`utils.autoattr.OneTimeProperty`

`utils.autoattr.ResetMixin`

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.

8.94.2 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.
```

8.94.3 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
```

8.95 utils.codeutil

8.95.1 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

8.95.2 Functions

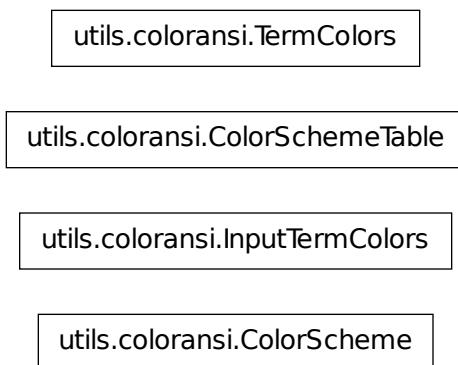
```
IPython.utils.codeutil.code_ctor(*args)
```

```
IPython.utils.codeutil.reduce_code(co)
```

8.96 utils.coloransi

8.96.1 Module: `utils.coloransi`

Inheritance diagram for `IPython.utils.coloransi`:



Tools for coloring text in ANSI terminals.

8.96.2 Classes

ColorScheme

```
class IPython.utils.coloransi.ColorScheme (_ColorScheme__scheme_name_, color-  
dict=None, **colormap)  
    Generic color scheme class. Just a name and a Struct.  
  
    __init__ (_ColorScheme__scheme_name_, colordict=None, **colormap)  
  
    copy (name=None)  
        Return a full copy of the object, optionally renaming it.
```

ColorSchemeTable

```
class IPython.utils.coloransi.ColorSchemeTable (scheme_list=None,           de-  
                                              fault_scheme='')  
    Bases: dict  
  
    General class to handle tables of color schemes.  
  
    It's basically a dict of color schemes with a couple of shorthand attributes and some convenient meth-  
ods.  
  
    active_scheme_name -> obvious active_colors -> actual color table of the active scheme  
  
    __init__ (scheme_list=None, default_scheme='')  
        Create a table of color schemes.  
  
        The table can be created empty and manually filled or it can be created with a list of valid color  
        schemes AND the specification for the default active scheme.  
  
    add_scheme (new_scheme)  
        Add a new color scheme to the table.  
  
    clear  
        D.clear() -> None. Remove all items from D.  
  
    copy()  
        Return full copy of object  
  
    static fromkeys (S[, v]) → New dict with keys from S and values equal to v.  
        v defaults to None.  
  
    get  
        D.get(k[,d]) -> D[k] if k in D, else d. d defaults to None.  
  
    has_key  
        D.has_key(k) -> True if D has a key k, else False  
  
    items  
        D.items() -> list of D's (key, value) pairs, as 2-tuples  
  
    iteritems  
        D.iteritems() -> an iterator over the (key, value) items of D
```

iterkeys

D.iterkeys() -> an iterator over the keys of D

itervalues

D.itervalues() -> an iterator over the values of D

keys

D.keys() -> list of D's keys

pop

D.pop(k[,d]) -> v, remove specified key and return the corresponding value. If key is not found, d is returned if given, otherwise KeyError is raised

popitem

D.popitem() -> (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

set_active_scheme (*scheme, case_sensitive=0*)

Set the currently active scheme.

Names are by default compared in a case-insensitive way, but this can be changed by setting the parameter *case_sensitive* to true.

setdefault

D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D

update

D.update(E, **F) -> None. Update D from dict/iterable E and F. If E has a .keys() method, does: for k in E: D[k] = E[k] If E lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values

D.values() -> list of D's values

InputTermColors

class IPython.utils.coloransi.InputTermColors

Color escape sequences for input prompts.

This class is similar to TermColors, but the escapes are wrapped in “ and ” so that readline can properly know the length of each line and can wrap lines accordingly. Use this class for any colored text which needs to be used in input prompts, such as in calls to raw_input().

This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a NoColor escape which is just the null string, suitable for defining ‘dummy’ color schemes in terminals which get confused by color escapes.

This class should be used as a mixin for building color schemes.

Black = '\x01\x1b[0;30m\x02'

BlinkBlack = '\x01\x1b[5;30m\x02'

BlinkBlue = '\x01\x1b[5;34m\x02'

```
BlinkCyan = '\x01\x1b[5;36m\x02'
BlinkGreen = '\x01\x1b[5;32m\x02'
BlinkLightGray = '\x01\x1b[5;37m\x02'
BlinkPurple = '\x01\x1b[5;35m\x02'
BlinkRed = '\x01\x1b[5;31m\x02'
BlinkYellow = '\x01\x1b[5;33m\x02'
Blue = '\x01\x1b[0;34m\x02'
Brown = '\x01\x1b[0;33m\x02'
Cyan = '\x01\x1b[0;36m\x02'
DarkGray = '\x01\x1b[1;30m\x02'
Green = '\x01\x1b[0;32m\x02'
LightBlue = '\x01\x1b[1;34m\x02'
LightCyan = '\x01\x1b[1;36m\x02'
LightGray = '\x01\x1b[0;37m\x02'
LightGreen = '\x01\x1b[1;32m\x02'
LightPurple = '\x01\x1b[1;35m\x02'
LightRed = '\x01\x1b[1;31m\x02'
NoColor = ''
Normal = '\x01\x1b[0m\x02'
Purple = '\x01\x1b[0;35m\x02'
Red = '\x01\x1b[0;31m\x02'
White = '\x01\x1b[1;37m\x02'
Yellow = '\x01\x1b[1;33m\x02'
```

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.

```
Black = '\x1b[0;30m'
BlinkBlack = '\x1b[5;30m'
```

```
BlinkBlue = '\x1b[5;34m'
BlinkCyan = '\x1b[5;36m'
BlinkGreen = '\x1b[5;32m'
BlinkLightGray = '\x1b[5;37m'
BlinkPurple = '\x1b[5;35m'
BlinkRed = '\x1b[5;31m'
BlinkYellow = '\x1b[5;33m'
Blue = '\x1b[0;34m'
Brown = '\x1b[0;33m'
Cyan = '\x1b[0;36m'
DarkGray = '\x1b[1;30m'
Green = '\x1b[0;32m'
LightBlue = '\x1b[1;34m'
LightCyan = '\x1b[1;36m'
LightGray = '\x1b[0;37m'
LightGreen = '\x1b[1;32m'
LightPurple = '\x1b[1;35m'
LightRed = '\x1b[1;31m'
NoColor =
Normal = '\x1b[0m'
Purple = '\x1b[0;35m'
Red = '\x1b[0;31m'
White = '\x1b[1;37m'
Yellow = '\x1b[1;33m'
```

8.96.3 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.

8.97 utils.daemonize

8.97.1 Module: utils.daemonize

daemonize function from twisted.scripts._twistd_unix.

```
IPython.utils.daemonize()
```

8.98 utils.data

8.98.1 Module: utils.data

Utilities for working with data structures like lists, dicts and tuples.

8.98.2 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.unique_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.

8.99 utils.decorators

8.99.1 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.

8.100 utils.dir2

8.100.1 Module: `utils.dir2`

A fancy version of Python’s builtin `dir()` function.

8.100.2 Functions

`IPython.utils.dir2.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)`

8.101 utils.doctestreload

8.101.1 Module: `utils.doctestreload`

A utility for handling the reloading of doctests.

8.101.2 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.

8.102 utils.frame

8.102.1 Module: `utils.frame`

Utilities for working with stack frames.

8.102.2 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.

8.103 utils.generics

8.103.1 Module: `utils.generics`

Generic functions for extending IPython.

See <http://cheeseshop.python.org/pypi/simplegeneric>.

8.103.2 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 :

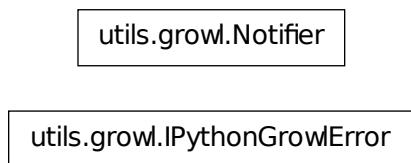
add to the attributes already discovered normally, return :

```
own_attrs + prev_completions. :  
IPython.utils.generics.inspect_object(*args, **kw)  
    Called when you do obj?
```

8.104 utils.growl

8.104.1 Module: utils.growl

Inheritance diagram for IPython.utils.growl:



Utilities using Growl on OS X for notifications.

8.104.2 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)
```

8.104.3 Functions

```
IPython.utils.growl.notify(title, msg)
IPython.utils.growl.notify_deferred(r, msg)
IPython.utils.growl.start(app_name)
```

8.105 utils.importstring

8.105.1 Module: `utils.importstring`

A simple utility to import something by its string name.

Authors:

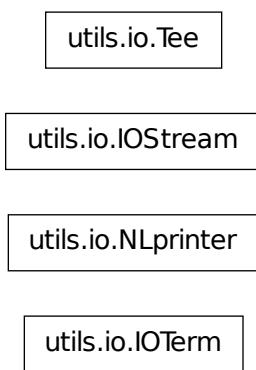
- Brian Granger

```
IPython.utils.importstring.import_item(name)
Import and return bar given the string foo.bar.
```

8.106 utils.io

8.106.1 Module: `utils.io`

Inheritance diagram for `IPython.utils.io`:



IO related utilities.

8.106.2 Classes

`IOStream`

```
class IPython.utils.io.IOStream(stream,fallback=None)
```

```
    __init__(stream,fallback=None)
    close()
    closed
    write(data)
    writelines(lines)
```

`IOTerm`

```
class IPython.utils.io.IOTerm(stdin=None, stdout=None, stderr=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__(stdin=None, stdout=None, stderr=None)
```

`NLprinter`

```
class IPython.utils.io.NLprinter
```

Print an arbitrarily nested list, indicating index numbers.

An instance of this class called nlprint is available and callable as a function.

nlprint(list,indent=' ',sep=': ') -> prints indenting each level by ‘indent’ and using ‘sep’ to separate the index from the value.

```
    __init__()
```

`Tee`

```
class IPython.utils.io.Tee(file_or_name, mode=None, channel='stdout')
```

Bases: object

A class to duplicate an output stream to stdout/err.

This works in a manner very similar to the Unix ‘tee’ command.

When the object is closed or deleted, it closes the original file given to it for duplication.

```
    __init__(file_or_name, mode=None, channel='stdout')
```

Construct a new Tee object.

Parameters `file_or_name` : filename or open filehandle (writable)

File that will be duplicated

mode : optional, valid mode for open().

If a filename was give, open with this mode.

channel : str, one of ['stdout', 'stderr']

close()

Close the file and restore the channel.

flush()

Flush both channels.

write(data)

Write data to both channels.

8.106.3 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.

8.107 utils.ipstruct

8.107.1 Module: `utils.ipstruct`

Inheritance diagram for IPython.utils.ipstruct:

```
utils.ipstruct.Struct
```

A dict subclass that supports attribute style access.

Authors:

- Fernando Perez (original)
- Brian Granger (refactoring to a dict subclass)

8.107.2 `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(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 reso-

lution 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

8.108 utils.jsonutil

8.108.1 Module: `utils.jsonutil`

Utilities to manipulate JSON objects.

8.108.2 Functions

`IPython.utils.jsonutil.date_default(obj)`
default function for packing datetime objects in JSON.

`IPython.utils.jsonutil.extract_dates(obj)`
extract ISO8601 dates from unpacked JSON

`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
```

`IPython.utils.jsonutil.rekey(dikt)`

Rekey a dict that has been forced to use str keys where there should be ints by json.

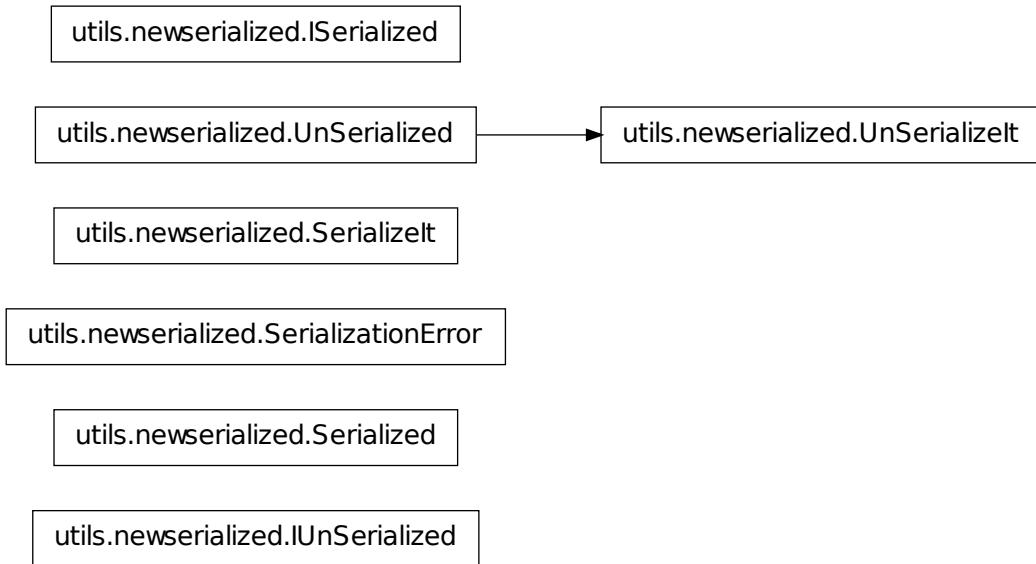
`IPython.utils.jsonutil.squash_dates(obj)`

squash datetime objects into ISO8601 strings

8.109 utils.newserialized

8.109.1 Module: `utils.newserialized`

Inheritance diagram for `IPython.utils.newserialized`:



Refactored serialization classes and interfaces.

8.109.2 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()
```

8.109.3 Functions

```
IPython.utils.newserialized.serialize(obj)
IPython.utils.newserialized.unserialize(serialized)
```

8.110 utils.notification

8.110.1 Module: utils.notification

Inheritance diagram for IPython.utils.notification:

```
utils.notification.NotificationCenter
```

```
utils.notification.NotificationError
```

The IPython Core Notification Center.

See docs/source/development/notification_blueprint.txt for an overview of the notification module.

Authors:

- Barry Wark
- Brian Granger

8.110.2 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 “`callback(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`

8.111 `utils.path`

8.111.1 Module: `utils.path`

Inheritance diagram for `IPython.utils.path`:

`utils.path.HomeDirError`

Utilities for path handling.

8.111.2 Class

8.111.3 `HomeDirError`

`class IPython.utils.path.HomeDirError`

Bases: `exceptions.Exception`

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`args`

`message`

8.111.4 Functions

`IPython.utils.path.check_for_old_config(ipython_dir=None)`

Check for old config files, and present a warning if they exist.

A link to the docs of the new config is included in the message.

This should mitigate confusion with the transition to the new config system in 0.11.

`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.isfile(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.filehash(path)`

Make an MD5 hash of a file, ignoring any differences in line ending characters.

`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%HOME% - %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 then 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.

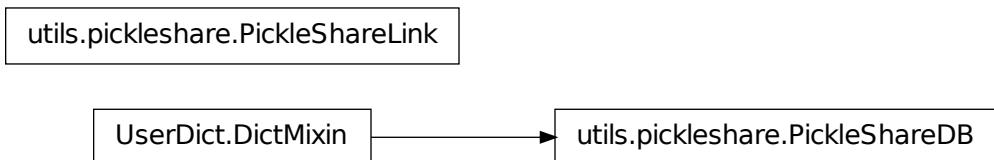
`target_update(target,deps,cmd) -> runs cmd if target is outdated.`

This is just a wrapper around `target_outdated()` which calls the given command if target is outdated.

8.112 utils.pickleshare

8.112.1 Module: `utils.pickleshare`

Inheritance diagram for `IPython.utils.pickleshare`:



PickleShare - a small ‘shelve’ like datastore with concurrency support

Like shelve, a `PickleShareDB` object acts like a normal dictionary. Unlike shelve, many processes can access the database simultaneously. Changing a value in database is immediately visible to other processes accessing the same database.

Concurrency is possible because the values are stored in separate files. Hence the “database” is a directory where *all* files are governed by `PickleShare`.

Example usage:

```
from pickleshare import *
db = PickleShareDB('~/testpickleshare')
db.clear()
print "Should be empty:", db.items()
db['hello'] = 15
db['aku ankka'] = [1, 2, 313]
db['paths/are/ok/key'] = [1, (5, 46)]
print db.keys()
del db['aku ankka']
```

This module is certainly not ZODB, but can be used for low-load (non-mission-critical) situations where tiny code size trumps the advanced features of a “real” object database.

Installation guide: `easy_install pickleshare`

Author: Ville Vainio <vivainio@gmail.com> License: MIT open source license.

8.112.2 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 specified 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 0x306a0f0>, fast_only=True)`
hashed get

`hset(hashroot, key, value)`
hashed set

`items()`

`iteritems()`

`iterkeys()`

`itervalues()`

`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)
```

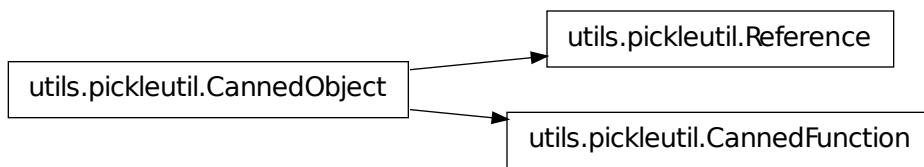
8.112.3 Functions

```
IPython.utils.pickleshare.gethashfile(key)
IPython.utils.pickleshare.main()
IPython.utils.pickleshare.stress()
IPython.utils.pickleshare.test()
```

8.113 utils.pickleutil

8.113.1 Module: utils.pickleutil

Inheritance diagram for IPython.utils.pickleutil:



Pickle related utilities. Perhaps this should be called ‘can’.

8.113.2 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)
```

8.113.3 Functions

```
IPython.utils.pickleutil.can(obj)
IPython.utils.pickleutil.canDict(obj)
IPython.utils.pickleutil.canSequence(obj)
IPython.utils.pickleutil.rebindFunctionGlobals(f, gbls)
IPython.utils.pickleutil.uncan(obj, g=None)
IPython.utils.pickleutil.uncanDict(obj, g=None)
IPython.utils.pickleutil.uncanSequence(obj, g=None)
```

8.114 utils.process

8.114.1 Module: `utils.process`

Inheritance diagram for IPython.utils.process:

```
utils.process.FindCmdError
```

Utilities for working with external processes.

8.114.2 Class

8.114.3 `FindCmdError`

```
class IPython.utils.process.FindCmdError
    Bases: exceptions.Exception

    __init__()
        x.__init__(...) initializes x; see x.__class__.__doc__ for signature

    args
    message
```

8.114.4 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. :

8.115 utils.strdispatch

8.115.1 Module: `utils.strdispatch`

Inheritance diagram for `IPython.utils.strdispatch`:

```
utils.strdispatch.StrDispatch
```

String dispatch class to match regexps and dispatch commands.

8.115.2 `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)
```

8.116 utils.sysinfo

8.116.1 Module: `utils.sysinfo`

Utilities for getting information about IPython and the system it's running in.

8.116.2 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.

8.117 utils.syspathcontext

8.117.1 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

8.117.2 Classes

`appended_to_syspath`

```
class IPython.utils.syspathcontext.appended_to_syspath(dir)
Bases: object
```

A context for appending a directory to sys.path for a second.

```
__init__(dir)
```

`prepended_to_syspath`

```
class IPython.utils.syspathcontext.prepended_to_syspath(dir)
Bases: object
```

A context for prepending a directory to sys.path for a second.

```
__init__(dir)
```

8.118 utils.terminal

8.118.1 Module: `utils.terminal`

Utilities for working with terminals.

Authors:

- Brian E. Granger
- Fernando Perez
- Alexander Belchenko (e-mail: bialix AT ukr.net)

8.118.2 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.

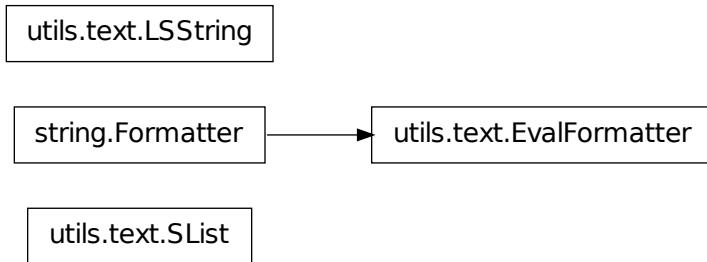
Parameters `val` : bool

If True, set_term_title() actually writes to the terminal (using the appropriate platform-specific module). If False, it is a no-op.

8.119 utils.text

8.119.1 Module: `utils.text`

Inheritance diagram for IPython.utils.text:



Utilities for working with strings and text.

8.119.2 Classes

`EvalFormatter`

class IPython.utils.text.**EvalFormatter**

Bases: `string.Formatter`

A String Formatter that allows evaluation of simple expressions.

Any time a format key is not found in the kwargs, it will be tried as an expression in the kwargs namespace.

This is to be used in templating cases, such as the parallel batch script templates, where simple arithmetic on arguments is useful.

Examples

In [1]: `f = EvalFormatter()` In [2]: `f.format('{n/4}', n=8)` Out[2]: '2'

In [3]: `f.format('{range(3)})')` Out[3]: '[0, 1, 2]'

In [4]: `f.format('{3*2})')` Out[4]: '6'

`__init__()`

`x.__init__(...)` initializes `x`; see `x.__class__.__doc__` for signature

`check_unused_args(used_args, args, kwargs)`

`convert_field(value, conversion)`

`format(format_string, *args, **kwargs)`

`format_field(value, format_spec)`

`get_field(field_name, args, kwargs)`

`get_value(key, args, kwargs)`

`parse(format_string)`

`vformat(format_string, args, kwargs)`

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) -> string

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(iterable) -> string

Return a string which is the concatenation of the strings in the iterable. 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) -> (head, sep, tail)

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

8.119.3 Functions

`IPython.utils.text.dedent` (*text*)

Equivalent of `textwrap.dedent` that ignores unindented first line.

This means it will still dedent strings like: `“foo is a bar”`

For use in `wrap_paragraphs`.

`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` (*instr*, *nspaces=4*, *ntabs=0*, *flatten=False*)

Indent a string a given number of spaces or tabstops.

`indent(str,nspaces=4,ntabs=0)` -> indent str by `ntabs+nspaces`.

Parameters `instr` : basestring

The string to be indented.

`nspaces` : int (default: 4)

The number of spaces to be indented.

ntabs : int (default: 0)

The number of tabs to be indented.

flatten : bool (default: False)

Whether to scrub existing indentation. If True, all lines will be aligned to the same indentation. If False, existing indentation will be strictly increased.

Returns `str|unicode` : string indented by ntabs and 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_lo1** ('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.

IPython.utils.text.**wrap_paragraphs** (text, ncols=80)
Wrap multiple paragraphs to fit a specified width.

This is equivalent to textwrap.wrap, but with support for multiple paragraphs, as separated by empty lines.

Returns list of complete paragraphs, wrapped to fill ‘ncols’ columns. :

8.120 utils.timing

8.120.1 Module: `utils.timing`

Utilities for timing code execution.

8.120.2 Functions

IPython.utils.timing.**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.

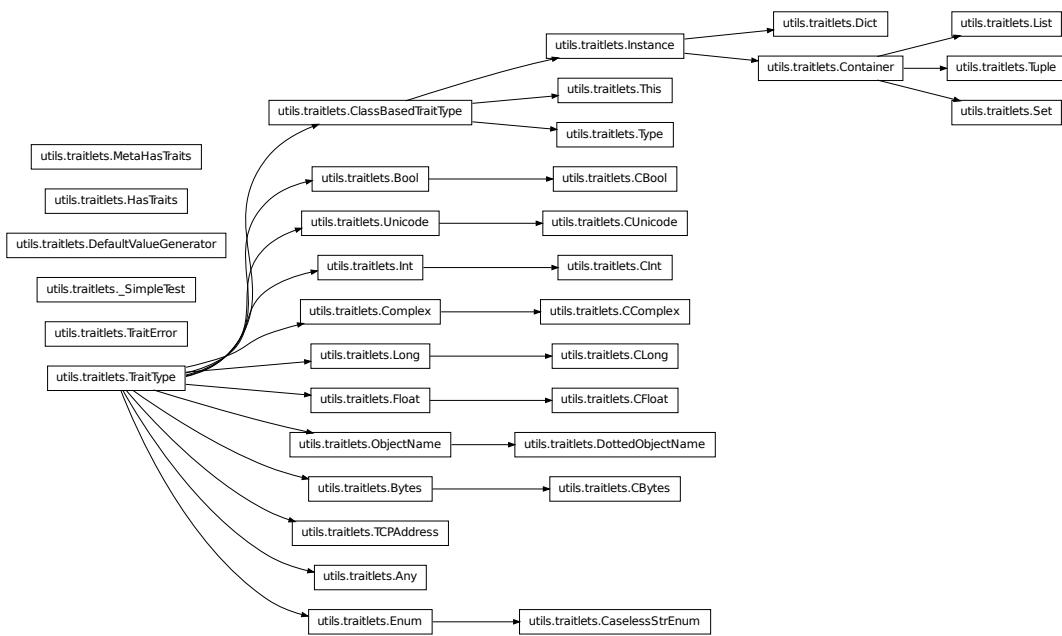
Under Unix, the return value is the sum of user+system time consumed by the process, computed via the resource module. This prevents problems related to the wraparound effect which the time.clock() function has.

Under Windows the return value is in wall clock seconds. See the documentation for the time module for more details.

8.121 utils.traits

8.121.1 Module: `utils.traits`

Inheritance diagram for `IPython.utils.traits`:



A lightweight Traits like module.

This is designed to provide a lightweight, simple, pure Python version of many of the capabilities of `enthought.traits`. This includes:

- Validation
- Type specification with defaults
- Static and dynamic notification
- Basic predefined types
- An API that is similar to `enthought.traits`

We don't support:

- Delegation
- Automatic GUI generation
- A full set of trait types. Most importantly, we don't provide container traits (list, dict, tuple) that can trigger notifications if their contents change.

- API compatibility with enthought.traits

There are also some important difference in our design:

- enthought.traits does not validate default values. We do.

We choose to create this module because we need these capabilities, but we need them to be pure Python so they work in all Python implementations, including Jython and IronPython.

Authors:

- Brian Granger
- Enthought, Inc. Some of the code in this file comes from enthought.traits and is licensed under the BSD license. Also, many of the ideas also come from enthought.traits even though our implementation is very different.

8.121.2 Classes

Any

```
class IPython.utils traitlets. Any (default_value=<IPython.utils traitlets.NoDefaultSpecified
                                     object at 0x3496250>, **metadata)
Bases: IPython.utils traitlets TraitType

__init__ (default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>,
          **metadata)
    Create a TraitType.

default_value = None

error (obj, value)

get_default_value ()
    Create a new instance of the default value.

get_metadata (key)

info ()

info_text = 'any value'

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.

```
metadata = {}

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)
```

Bool

```
class IPython.utils.traits.Bool(default_value=<IPython.utils.traits.NoDefaultSpecified
                                  object at 0x3496250>, **metadata)
Bases: IPython.utils.traits.TraitType
```

A boolean (True, False) trait.

```
__init__(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
        **metadata)
    Create a TraitType.
```

```
default_value = False
```

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a boolean'
```

```
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.

```
metadata = {}
```

```
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)
```

Bytes

```
class IPython.utils traitlets.Bytes (default_value=<IPython.utils traitlets.NoDefaultSpecified
object at 0x3496250>, **metadata)
Bases: IPython.utils traitlets TraitType
```

A trait for strings.

```
__init__(default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>,
         **metadata)
Create a TraitType.
```

```
default_value = ''
```

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a string'
```

```
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.

```
metadata = {}
```

```
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)
```

CBool

```
class IPython.utils traitlets.CBool (default_value=<IPython.utils traitlets.NoDefaultSpecified
object at 0x3496250>, **metadata)
Bases: IPython.utils traitlets.Bool
```

A casting version of the boolean trait.

```
__init__(default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>,
         **metadata)
Create a TraitType.
```

```
default_value = False
```

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a boolean'
```

```
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.

```
metadata = {}
```

```
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)
```

CBytes

```
class IPython.utils.traits.CBytes (default_value=<IPython.utils.traits.NoDefaultSpecified
object at 0x3496250>, **metadata)
Bases: IPython.utils.traits.Bytes
```

A casting version of the string trait.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
          **metadata)
Create a TraitType.
```

```
default_value = ''
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a string'
```

```
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.

```
metadata = {}
```

```
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)
```

CComplex

```
class IPython.utils.traits.CComplex(default_value=<IPython.utils.traits.NoDefaultSpecified
                                     object at 0x3496250>, **metadata)
Bases: IPython.utils.traits.Complex
```

A casting version of the complex number trait.

```
__init__(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
         **metadata)
Create a TraitType.
```

```
default_value = 0j
```

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a complex number'
```

```
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.

```
metadata = {}
```

```
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)
```

CFloat

```
class IPython.utils.traits.CFloat (default_value=<IPython.utils.traits.NoDefaultSpecified
object at 0x3496250>, **metadata)
Bases: IPython.utils.traits.Float
```

A casting version of the float trait.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
          **metadata)
Create a TraitType.
```

```
default_value = 0.0
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a float'
```

```
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.

```
metadata = {}
```

```
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)
```

CInt

class IPython.utils.traits.CInt (*default_value*=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, ***metadata*)
Bases: IPython.utils.traits.Int

A casting version of the int trait.

__init__ (*default_value*=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, ***metadata*)
Create a TraitType.

default_value = 0

error (*obj*, *value*)

get_default_value ()

Create a new instance of the default value.

get_metadata (*key*)

info ()

info_text = ‘an integer’

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.

metadata = {}

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*)

CLong

```
class IPython.utils.traits.CLong (default_value=<IPython.utils.traits.NoDefaultSpecified  
object at 0x3496250>, **metadata)  
Bases: IPython.utils.traits.Long
```

A casting version of the long integer trait.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,  
          **metadata)  
    Create a TraitType.
```

```
default_value = 0L
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a long'
```

```
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.

```
metadata = {}
```

```
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)
```

CUnicode

```
class IPython.utils.traits.CUnicode (default_value=<IPython.utils.traits.NoDefaultSpecified  
object at 0x3496250>, **metadata)  
Bases: IPython.utils.traits.Unicode
```

A casting version of the unicode trait.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,  
          **metadata)  
    Create a TraitType.
```

```
default_value = u''
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a unicode string'
```

```
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.

```
metadata = {}
```

```
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)
```

CaselessStrEnum

```
class IPython.utils.traits.CaselessStrEnum(values, default_value=None, allow_none=True, **metadata)
```

Bases: [IPython.utils.traits.Enum](#)

An enum of strings that are caseless in validate.

```
__init__(values, default_value=None, allow_none=True, **metadata)
```

default_value = <IPython.utils.traits.Undefined object at 0x34962d0>

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

Returns a description of the trait.

```
info_text = 'any value'
```

```
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.

```
metadata = {}
```

```
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)
```

ClassBasedTraitType

```
class IPython.utils.traits.ClassBasedTraitType (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata)
```

Bases: [IPython.utils.traits.TraitType](#)

A trait with error reporting for Type, Instance and This.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata)
```

Create a TraitType.

```
default_value = <IPython.utils.traits.Undefined object at 0x34962d0>
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'any value'
```

```
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.

```
metadata = {}
```

```
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)
```

Complex

```
class IPython.utils.traits.Complex(default_value=<IPython.utils.traits.NoDefaultSpecified
object at 0x3496250>, **metadata)
Bases: IPython.utils.traits.TraitType
```

A trait for complex numbers.

```
__init__(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
         **metadata)
Create a TraitType.
```

```
default_value = 0j
```

```
error(obj, value)
```

```
get_default_value()
```

Create a new instance of the default value.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a complex number'
```

```
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.

```
metadata = {}
```

```
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)
```

Container

```
class IPython.utils.traits.Container(trait=None, default_value=None, allow_none=True, **metadata)
Bases: IPython.utils.traits.Instance
```

An instance of a container (list, set, etc.)

To be subclassed by overriding klass.

```
__init__(trait=None, default_value=None, allow_none=True, **metadata)
```

Create a container trait type from a list, set, or tuple.

The default value is created by doing `List(default_value)`, which creates a copy of the `default_value`.

`trait` can be specified, which restricts the type of elements in the container to that TraitType.

If only one arg is given and it is not a Trait, it is taken as `default_value`:

```
c = List([1, 2, 3])
```

Parameters `trait` : TraitType [optional]

the type for restricting the contents of the Container. If unspecified, types are not checked.

`default_value` : SequenceType [optional]

The default value for the Trait. Must be list/tuple/set, and will be cast to the container type.

`allow_none` : Bool [default True]

Whether to allow the value to be None

`**metadata` : any

further keys for extensions to the Trait (e.g. config)

```
default_value = <IPython.utils.traits.Undefined object at 0x34962d0>
```

```
element_error(obj, element, validator)
```

```
error(obj, value)
```

```
get_default_value()
```

Instantiate a default value instance.

This is called when the containing HasTraits classes' `__new__()` method is called to ensure that a unique instance is created for each HasTraits instance.

```
get_metadata(key)
```

```
info()
```

```
info_text = 'any value'
```

```
init()
```

```
instance_init (obj)
klass = None
metadata = {}
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)
validate_elements (obj, value)
```

DefaultValueGenerator

```
class IPython.utils traitlets.DefaultValueGenerator (*args, **kw)
Bases: object
```

A class for generating new default value instances.

```
__init__ (*args, **kw)
generate (klass)
```

Dict

```
class IPython.utils traitlets.Dict (default_value=None, allow_none=True, **metadata)
Bases: IPython.utils traitlets.Instance
```

An instance of a Python dict.

```
__init__ (default_value=None, allow_none=True, **metadata)
Create a dict trait type from a dict.
```

The default value is created by doing `dict(default_value)`, which creates a copy of the `default_value`.

```
default_value = <IPython.utils traitlets.Undefined object at 0x34962d0>
```

```
error (obj, value)
```

```
get_default_value ()
```

Instantiate a default value instance.

This is called when the containing `HasTraits` classes' `__new__()` method is called to ensure that a unique instance is created for each `HasTraits` instance.

```
get_metadata (key)
```

```
info()
info_text = 'any value'

init()

instance_init(obj)

metadata = {}

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)
```

DottedObjectName

```
class IPython.utils.traits.DottedObjectName(default_value=<IPython.utils.traits.NoDefaultSpecified
                                             object at 0x3496250>, **meta-
                                             data)
```

Bases: `IPython.utils.traits.ObjectName`

A string holding a valid dotted object name in Python, such as `A.b3._c`

```
__init__(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
         **metadata)
Create a TraitType.
```

```
coerce_str(obj, value)
In Python 2, coerce ascii-only unicode to str
```

```
default_value = <IPython.utils.traits.Undefined object at 0x34962d0>
```

```
error(obj, value)
```

```
get_default_value()
Create a new instance of the default value.
```

```
get_metadata(key)
```

```
info()
```

```
info_text = 'a valid object identifier in Python'
```

```
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.

`isidentifier(s)`

`metadata = {}`

`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)`

Enum

`class IPython.utils traitlets.Enum(values, default_value=None, allow_none=True, **metadata)`

Bases: `IPython.utils traitlets TraitType`

An enum that whose value must be in a given sequence.

`__init__(values, default_value=None, allow_none=True, **metadata)`

`default_value = <IPython.utils traitlets Undefined object at 0x34962d0>`

`error(obj, value)`

`get_default_value()`

Create a new instance of the default value.

`get_metadata(key)`

`info()`

Returns a description of the trait.

`info_text = ‘any value’`

`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.

`metadata = {}`

`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)`

`Float`

`class IPython.utils.traits.Float(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata)`

Bases: `IPython.utils.traits.TraitType`

A float trait.

`__init__(default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata)`

Create a TraitType.

`default_value = 0.0`

`error(obj, value)`

`get_default_value()`

Create a new instance of the default value.

`get_metadata(key)`

`info()`

`info_text = 'a float'`

`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.

```
metadata = {}

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)
```

HasTraits

```
class IPython.utils traitlets. HasTraits (**kw)
    Bases: object
```

```
__init__(**kw)
```

```
classmethod class_trait_names(**metadata)
```

Get a list of all the names of this classes traits.

This method is just like the `trait_names()` method, but is unbound.

```
classmethod class_traits(**metadata)
```

Get a list of all the traits of this class.

This method is just like the `traits()` method, but is unbound.

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.

```
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.

Instance

class IPython.utils.traits.**Instance** (*klass=None*, *args=None*, *kw=None*, *allow_none=True*, ***metadata*)

Bases: IPython.utils.traits.ClassBasedTraitType

A trait 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__ (*klass=None*, *args=None*, *kw=None*, *allow_none=True*, ***metadata*)

Construct an Instance trait.

This trait allows values that are instances of a particular class or its subclasses. Our implementation is quite different from that of enthough.traits as we don't allow instances to be used for klass and we handle the args and kw arguments differently.

Parameters **klass** : class, str

The class that forms the basis for the trait. Class names can also be specified as strings, like 'foo.bar.Bar'.

args : tuple

Positional arguments for generating the default value.

kw : dict

Keyword arguments for generating the default value.

allow_none : bool

Indicates whether None is allowed as a value.

default_value = <IPython.utils.traits.Undefined object at 0x34962d0>

error (*obj*, *value*)

get_default_value()

Instantiate a default value instance.

This is called when the containing HasTraits classes' `__new__()` method is called to ensure that a unique instance is created for each HasTraits instance.

get_metadata(key)**info()****info_text = 'any value'****init()****instance_init(obj)****metadata = {}****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)****Int**

class IPython.utils traitlets.Int (default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>, **metadata)

Bases: `IPython.utils traitlets TraitType`

A integer trait.

__init__(default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>, **metadata)

Create a TraitType.

default_value = 0**error(obj, value)****get_default_value()**

Create a new instance of the default value.

get_metadata(key)**info()****info_text = 'an integer'****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.

metadata = {}**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*)**List**

```
class IPython.utils.traitlets.List(trait=None, default_value=None, minlen=0,
                                    maxlen=9223372036854775807, al-
                                    low_none=True, **metadata)
```

Bases: `IPython.utils.traitlets.Container`

An instance of a Python list.

```
__init__(trait=None, default_value=None, minlen=0, maxlen=9223372036854775807, al-
         low_none=True, **metadata)
```

Create a List trait type from a list, set, or tuple.

The default value is created by doing `List(default_value)`, which creates a copy of the `default_value`.

`trait` can be specified, which restricts the type of elements in the container to that TraitType.

If only one arg is given and it is not a Trait, it is taken as `default_value`:

```
c = List([1, 2, 3])
```

Parameters `trait` : TraitType [optional]

the type for restricting the contents of the Container. If unspecified, types are not checked.

default_value : SequenceType [optional]

The default value for the Trait. Must be list/tuple/set, and will be cast to the container type.

minlen : Int [default 0]

The minimum length of the input list

maxlen : Int [default sys.maxint]

The maximum length of the input list

allow_none : Bool [default True]

Whether to allow the value to be None

****metadata** : any

further keys for extensions to the Trait (e.g. config)

default_value = <IPython.utils.traits.Undefined object at 0x34962d0>

element_error (obj, element, validator)

error (obj, value)

get_default_value ()

Instantiate a default value instance.

This is called when the containing HasTraits classes' `__new__()` method is called to ensure that a unique instance is created for each HasTraits instance.

get_metadata (key)

info ()

info_text = 'any value'

init ()

instance_init (obj)

klass

alias of list

length_error (obj, value)

metadata = {}

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)

validate_elements (obj, value)

Long

```
class IPython.utils.traits.Long (default_value=<IPython.utils.traits.NoDefaultSpecified  
object at 0x3496250>, **metadata)  
Bases: IPython.utils.traits.TraitType
```

A long integer trait.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,  
          **metadata)  
    Create a TraitType.
```

```
default_value = 0L
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'a long'
```

```
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.

```
metadata = {}
```

```
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)
```

MetaHasTraits**class** IPython.utils.traits.**MetaHasTraits** (*name, bases, classdict*)

Bases: type

A metaclass for HasTraits.

This metaclass makes sure that any TraitType class attributes are instantiated and sets their name attribute.

__init__ (*name, bases, classdict*)

Finish initializing the HasTraits class.

This sets the `this_class` attribute of each TraitType in the class dict to the newly created class `cls`.**mro**

mro() -> list return a type's method resolution order

NoDefaultSpecifiedIPython.utils.traits.**NoDefaultSpecified****ObjectName****class** IPython.utils.traits.**ObjectName** (*default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata*)

Bases: IPython.utils.traits.TraitType

A string holding a valid object name in this version of Python.

This does not check that the name exists in any scope.

__init__ (*default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>, **metadata*)

Create a TraitType.

coerce_str (*obj, value*)

In Python 2, coerce ascii-only unicode to str

default_value = <IPython.utils.traits.Undefined object at 0x34962d0>**error** (*obj, value*)**get_default_value** ()

Create a new instance of the default value.

get_metadata (*key*)**info** ()**info_text** = 'a valid object identifier in Python'**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.

isidentifier(*s*)**metadata** = {}**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*)**Set**

class `IPython.utils.traits.Set` (*trait=None, default_value=None, allow_none=True, **metadata*)

Bases: `IPython.utils.traits.Container`

An instance of a Python set.

__init__(*trait=None, default_value=None, allow_none=True, **metadata*)

Create a container trait type from a list, set, or tuple.

The default value is created by doing `List(default_value)`, which creates a copy of the `default_value`.

`trait` can be specified, which restricts the type of elements in the container to that TraitType.

If only one arg is given and it is not a Trait, it is taken as `default_value`:

```
c = List([1, 2, 3])
```

Parameters `trait` : TraitType [optional]

the type for restricting the contents of the Container. If unspecified, types are not checked.

default_value : SequenceType [optional]

The default value for the Trait. Must be list/tuple/set, and will be cast to the container type.

allow_none : Bool [default True]

Whether to allow the value to be None

****metadata** : any

further keys for extensions to the Trait (e.g. config)

default_value = <IPython.utils traitlets Undefined object at 0x34962d0>

element_error (obj, element, validator)

error (obj, value)

get_default_value ()

Instantiate a default value instance.

This is called when the containing HasTraits classes' `__new__()` method is called to ensure that a unique instance is created for each HasTraits instance.

get_metadata (key)

info ()

info_text = ‘any value’

init ()

instance_init (obj)

klass

alias of set

metadata = {}

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)

validate_elements (obj, value)

TCPAddress

class IPython.utils traitlets **TCPAddress** (`default_value=<IPython.utils traitlets.NoDefaultSpecified object at 0x3496250>, **metadata`)

Bases: IPython.utils traitlets TraitType

A trait for an (ip, port) tuple.

This allows for both IPv4 IP addresses as well as hostnames.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,
           **metadata)
    Create a TraitType.

default_value = ('127.0.0.1', 0)

error (obj, value)

get_default_value ()
    Create a new instance of the default value.

get_metadata (key)

info ()

info_text = 'an (ip, port) tuple'

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.

```
metadata = {}

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)
```

This

```
class IPython.utils.traits.This (**metadata)
Bases: IPython.utils.traits.ClassBasedTraitType
```

A trait for instances of the class containing this trait.

Because how how and when class bodies are executed, the `This` trait can only have a default value of `None`. This, and because we always validate default values, `allow_none` is *always* true.

```
__init__(**metadata)
default_value = <IPython.utils traitlets.Undefined object at 0x34962d0>
error(obj, value)
get_default_value()
    Create a new instance of the default value.
get_metadata(key)
info()
info_text = ‘an instance of the same type as the receiver or None’
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.
metadata = {}
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)

TraitError

class IPython.utils traitlets.TraitError
    Bases: exceptions.Exception
__init__()
    x.__init__(...) initializes x; see x.__class__.__doc__ for signature
args
message
```

TraitType

```
class IPython.utils.traits.TraitType (default_value=<IPython.utils.traits.NoDefaultSpecified  
object at 0x3496250>, **metadata)
```

Bases: object

A base class for all trait descriptors.

Notes

Our implementation of traits is based on Python's descriptor protocol. This class is the base class for all such descriptors. The only magic we use is a custom metaclass for the main `HasTraits` class that does the following:

- 1.Sets the `name` attribute of every `TraitType` instance in the class dict to the name of the attribute.

- 2.Sets the `this_class` attribute of every `TraitType` instance in the class dict to the `class` that declared the trait. This is used by the `This` trait to allow subclasses to accept superclasses for `This` values.

```
__init__ (default_value=<IPython.utils.traits.NoDefaultSpecified object at 0x3496250>,  
          **metadata)
```

Create a `TraitType`.

```
default_value = <IPython.utils.traits.Undefined object at 0x34962d0>
```

```
error (obj, value)
```

```
get_default_value ()
```

Create a new instance of the default value.

```
get_metadata (key)
```

```
info ()
```

```
info_text = 'any value'
```

```
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.

```
metadata = {}
```

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*)**Tuple****class IPython.utils traitlets.Tuple(*traits, **metadata)**

Bases: `IPython.utils traitlets.Container`

An instance of a Python tuple.

__init__(*traits, default_value=None, allow_none=True, **metadata)

Create a tuple from a list, set, or tuple.

Create a fixed-type tuple with Traits:

```
t = Tuple(Int, Str, CStr)
```

would be length 3, with Int,Str,CStr for each element.

If only one arg is given and it is not a Trait, it is taken as default_value:

```
t = Tuple((1, 2, 3))
```

Otherwise, `default_value` *must* be specified by keyword.

Parameters *traits : TraitTypes [optional]

the tsype for restricting the contents of the Tuple. If unspecified, types are not checked. If specified, then each positional argument corresponds to an element of the tuple. Tuples defined with traits are of fixed length.

default_value : SequenceType [optional]

The default value for the Tuple. Must be list/tuple/set, and will be cast to a tuple. If `traits` are specified, the `default_value` must conform to the shape and type they specify.

allow_none : Bool [default True]

Whether to allow the value to be None

****metadata : any**

further keys for extensions to the Trait (e.g. config)

default_value = <IPython.utils traitlets Undefined object at 0x34962d0>**element_error**(*obj, element, validator*)**error**(*obj, value*)

get_default_value()

Instantiate a default value instance.

This is called when the containing HasTraits classes' `__new__()` method is called to ensure that a unique instance is created for each HasTraits instance.

get_metadata(key)**info()****info_text = 'any value'****init()****instance_init(obj)****klass**

alias of tuple

metadata = {}**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)****validate_elements(obj, value)****Type**

```
class IPython.utils traitlets.Type (default_value=None,           klass=None,           al-
                                         low_none=True, **metadata)
Bases: IPython.utils traitlets ClassBasedTraitType
```

A trait whose value must be a subclass of a specified class.

__init__(default_value=None, klass=None, allow_none=True, **metadata)

Construct a Type trait

A Type trait specifies that its values must be subclasses of a particular class.

If only `default_value` is given, it is used for the `klass` as well.

Parameters `default_value` : class, str or None

The default value must be a subclass of `klass`. If an str, the str must be a fully specified class name, like 'foo.bar.Bah'. The string is resolved into real class, when the parent `HasTraits` class is instantiated.

`klass` : class, str, None

Values of this trait must be a subclass of klass. The klass may be specified in a string like: ‘foo.bar.MyClass’. The string is resolved into real class, when the parent `HasTraits` class is instantiated.

allow_none : boolean

Indicates whether `None` is allowed as an assignable value. Even if `False`, the default value may be `None`.

default_value = <`IPython.utils traitlets.Undefined` object at `0x34962d0`>

error (*obj, value*)

get_default_value ()

get_metadata (*key*)

info ()

Returns a description of the trait.

info_text = ‘any value’

init ()

instance_init (*obj*)

metadata = {}

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*)

Validates that the value is a valid object instance.

Undefined

`IPython.utils traitlets.Undefined`

Unicode

class `IPython.utils traitlets.Unicode` (*default_value*=<`IPython.utils traitlets.NoDefaultSpecified` object at `0x3496250`>, ***metadata*)
Bases: `IPython.utils traitlets TraitType`

A trait for unicode strings.

__init__ (*default_value*=<`IPython.utils traitlets.NoDefaultSpecified` object at `0x3496250`>, ***metadata*)
Create a TraitType.

```
default_value = u''

error(obj, value)

get_default_value()
    Create a new instance of the default value.

get_metadata(key)

info()

info_text = 'a unicode string'

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.

metadata = {}

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)
```

8.121.3 Functions

```
IPython.utils traitlets.add_article(name)
    Returns a string containing the correct indefinite article ('a' or 'an') prefixed to the specified string.

IPython.utils traitlets.class_of(object)
    Returns a string containing the class name of an object with the correct indefinite article ('a' or 'an') preceding it (e.g., 'an Image', 'a PlotValue').

IPython.utils traitlets.getmembers(object, predicate=None)
    A safe version of inspect.getmembers that handles missing attributes.

    This is useful when there are descriptor based attributes that for some reason raise AttributeError even though they exist. This happens in zope.interface with the __provides__ attribute.
```

`IPython.utils.traits.parse_notifier_name(name)`
Convert the name argument to a list of names.

Examples

```
>>> parse_notifier_name('a')
['a']
>>> parse_notifier_name(['a', 'b'])
['a', 'b']
>>> parse_notifier_name(None)
['anytrait']
```

`IPython.utils.traits.repr_type(obj)`
Return a string representation of a value and its type for readable error messages.

8.122 utils.upgradedir

8.122.1 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.

8.122.2 Functions

`IPython.utils.upgradedir.showdiff(old, new)`

`IPython.utils.upgradedir.upgrade_dir(srkdir, tgtdir)`
Copy over all files in srkdir to tgtdir w/ native line endings

Creates .upgrade_report in tgtdir that stores md5sums of all files to notice changed files b/w upgrades.

8.123 utils.warn

8.123.1 Module: `utils.warn`

Utilities for warnings. Shouldn’t we just use the built in warnings module.

8.123.2 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 `io.stderr` (`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.

8.124 utils.wildcard

8.124.1 Module: `utils.wildcard`

Support for wildcard pattern matching in object inspection.

Authors

- Jörgen Stenarson <jorgen.stenarson@bostream.nu>
- Thomas Kluyver

8.124.2 Functions

`IPython.utils.wildcard.create_typestr2type_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.

ABOUT IPYTHON

9.1 Credits

IPython was started and continues to be led by Fernando Pérez.

9.1.1 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.
- **Thomas Kluyver** <takowl-AT-gmail.com> Port of IPython and its necessary ZeroMQ infrastructure to Python3, IPython core.
- **Evan Patterson** <epatters-AT-enthought.com> Qt console frontend with ZeroMQ.

9.1.2 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>.

9.1.3 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.enthought.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.

9.1.4 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.
- 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 <wardefar-AT-iro.umontreal.ca> - bugfixes to %timeit, input autoindent management, and Qt console tooltips.
- 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 Ricupi <andrea.ricupi-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 idiosyncrasies 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 <retyp-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 Dodd <cdodd-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 <rhh-AT-structurelabs.com> Bug reports and fixes to logging module.
- Daniel ‘Dang’ Griffith <pythondev-dang-AT-lazytwinacres.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-hetdigitalegat.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 IPyhton, 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-initab.de> help with Debian packaging and distribution.
- George Sakkis <gsakkis-AT-eden.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-m11.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.

9.2 History

9.2.1 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.

9.3 License and Copyright

9.3.1 License

IPython is licensed under the terms of the new or revised BSD license, as follows:

Copyright (c) 2011, 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.

9.3.2 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

- Thomas Kluyver
- Brian E. Granger
- Evan Patterson
- 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.

9.3.3 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.

9.3.4 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

- [ZeroMQ] ZeroMQ. <http://www.zeromq.org>
- [paramiko] paramiko. <https://github.com/robey/paramiko>
- [pygments] Pygments syntax highlighting. <http://pygments.org>
- [pexpect] Pexpect. <http://www.noah.org/wiki/Pexpect>
- [PyQt] PyQt4 <http://www.riverbankcomputing.co.uk/software/pyqt/download>
- [pygments] Pygments <http://pygments.org/>
- [ZeroMQ] ZeroMQ. <http://www.zeromq.org>
- [MongoDB] MongoDB database <http://www.mongodb.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/>>
- [HMAC] <<http://tools.ietf.org/html/rfc2104.html>>
- [Emacs] Emacs. <http://www.gnu.org/software/emacs/>
- [TextMate] TextMate: the missing editor. <http://macromates.com/>
- [vim] vim. <http://www.vim.org/>
- [Git] The Git version control system.
- [Github.com] Github.com. <http://github.com>

[PEP8] Python Enhancement Proposal 8. <http://www.python.org/peps/pep-0008.html>

[reStructuredText] reStructuredText. <http://docutils.sourceforge.net/rst.html>

[Sphinx] Sphinx. <http://sphinx.pocoo.org/>

[MatplotlibDocGuide] http://matplotlib.sourceforge.net-devel/documenting_mpl.html

[PEP257] PEP 257. <http://www.python.org/peps/pep-0257.html>

[NumPyDocGuide] NumPy documentation guide. <http://projects.scipy.org/numpy/wiki/CodingStyleGuidelines>

[NumPyExampleDocstring] NumPy example docstring. http://projects.scipy.org/numpy/browser/trunk/doc/EXAMPLE_DOCSTRING

PYTHON MODULE INDEX

i

IPython.config.application, 263
IPython.config.configurable, 268
IPython.config.loader, 276
IPython.core.alias, 283
IPython.core.application, 287
IPython.core.autocall, 292
IPython.core.builtin_trap, 294
IPython.core.compilerop, 296
IPython.core.completer, 297
IPython.core.completerlib, 303
IPython.core.crashhandler, 304
IPython.core.debugger, 306
IPython.core.display, 315
IPython.core.display_trap, 316
IPython.core.displayhook, 318
IPython.core.displaypub, 322
IPython.core.error, 326
IPython.core.excolors, 327
IPython.core.extensions, 328
IPython.core.formatters, 330
IPython.core.history, 358
IPython.core.hooks, 366
IPython.core.inputsplitter, 368
IPython.core.interactiveshell, 375
IPython.core.ipapi, 417
IPython.core.logger, 417
IPython.core.macro, 419
IPython.core.magic, 420
IPython.core.magic_arguments, 444
IPython.core.oinspect, 448
IPython.core.page, 452
IPython.core.payload, 453
IPython.core.payloadpage, 456
IPython.core.plugin, 457
IPython.core.prefilter, 461

IPython.core.profileapp, 512
IPython.core.profiledir, 525
IPython.core.prompts, 529
IPython.core.shellapp, 532
IPython.core.splitinput, 535
IPython.core.ultratb, 536
IPython.lib.backgroundjobs, 547
IPython.lib.clipboard, 553
IPython.lib.deepreload, 553
IPython.lib.demo, 554
IPython.lib.guisupport, 568
IPython.lib.inputhook, 570
IPython.lib.irunner, 573
IPython.lib.latextools, 579
IPython.lib.pretty, 580
IPython.lib.pylabtools, 585
IPython.parallel.apps.baseapp, 587
IPython.parallel.apps.ipclusterapp,
593
IPython.parallel.apps.ipcontrollerapp,
615
IPython.parallel.apps.ipengineapp,
621
IPython.parallel.apps.iploggerapp,
630
IPython.parallel.apps.launcher, 636
IPython.parallel.apps.logwatcher,
717
IPython.parallel.apps.win32support,
720
IPython.parallel.apps.winhpcjob, 721
IPython.parallel.client.asyncresult,
739
IPython.parallel.client.client, 743
IPython.parallel.client.map, 752
IPython.parallel.client.remotefunction,
754

IPython.parallel.client.view, 756
IPython.parallel.controller.dependency, 772
IPython.parallel.controller.dictdb, 775
IPython.parallel.controller.heartmonitor, 781
IPython.parallel.controller.hub, 784
IPython.parallel.controller.scheduler, 794
IPython.parallel.controller.sqlitedb, 800
IPython.parallel.engine.engine, 803
IPython.parallel.engine.kernelstarter, 806
IPython.parallel.engine.streamkernel, 808
IPython.parallel.error, 812
IPython.parallel.factory, 823
IPython.parallel.util, 826
IPython.testing, 829
IPython.testing.decorators, 830
IPython.testing.globalipapp, 832
IPython.testing.ptest, 835
IPython.testing.ipunittest, 836
IPython.testing.mkdoctests, 838
IPython.testing.nosepatch, 840
IPython.testing.plugin.dtexample, 840
IPython.testing.plugin.show_refs, 842
IPython.testing.plugin.simple, 843
IPython.testing.plugin.test_ipdoctest, 843
IPython.testing.plugin.test_refs, 844
IPython.testing.skipdoctest, 845
IPython.testing.tools, 845
IPython.utils.attic, 850
IPython.utils.autoattr, 851
IPython.utils.codeutil, 854
IPython.utils.coloransi, 854
IPython.utils.daemonize, 859
IPython.utils.data, 859
IPython.utils.decorators, 860
IPython.utils.dir2, 860
IPython.utils.doctestreload, 860
IPython.utils.frame, 861
IPython.utils.generics, 862
IPython.utils.growl, 863
IPython.utils.importstring, 864
IPython.utils.io, 864
IPython.utils.ipstruct, 867
IPython.utils.jsonutil, 871
IPython.utils.newserialized, 873
IPython.utils.notification, 875
IPython.utils.path, 877
IPython.utils.pickleutil, 882
IPython.utils.process, 884
IPython.utils.PyColorize, 884
IPython.utils.strdispatch, 885
IPython.utils.sysinfo, 886
IPython.utils.syspathcontext, 887
IPython.utils.terminal, 888
IPython.utils.text, 889
IPython.utils.timing, 899
IPython.utils.traits, 900
IPython.utils.upgradedir, 935
IPython.utils.warn, 935
IPython.utils.wildcard, 936

INDEX

Symbols

| | |
|---|---|
| %PATH%, 140 | <code>__init__(IPython.config.loader.PyFileConfigLoader method)</code> , 283 |
| <code>__init__(IPython.config.application.Application method)</code> , 263 | <code>__init__(IPython.core.alias.AliasError method)</code> , 284 |
| <code>__init__(IPython.config.application.ApplicationError method)</code> , 267 | <code>__init__(IPython.core.alias.AliasManager method)</code> , 284 |
| <code>__init__(IPython.config.configurable.Configurable method)</code> , 268 | <code>__init__(IPython.core.alias.InvalidAliasError method)</code> , 286 |
| <code>__init__(IPython.config.configurable.ConfigurableError method)</code> , 270 | <code>__init__(IPython.core.application.BaseIPythonApplication method)</code> , 287 |
| <code>__init__(IPython.config.configurable.LoggingConfigurable method)</code> , 271 | <code>__init__(IPython.core.autocall.ExitAutocall method)</code> , 293 |
| <code>__init__(IPython.config.configurable.MultipleInstanceError method)</code> , 272 | <code>__init__(IPython.core.autocall.IPyAutocall method)</code> , 293 |
| <code>__init__(IPython.config.configurable.SingletonConfigurable method)</code> , 273 | <code>__init__(IPython.core.autocall.ZMQExitAutocall method)</code> , 293 |
| <code>__init__(IPython.config.loader.ArgParseConfigLoader method)</code> , 276 | <code>__init__(IPython.core.builtin_trap.BuiltinTrap method)</code> , 294 |
| <code>__init__(IPython.config.loader.ArgumentParser method)</code> , 277 | <code>__init__(IPython.core.compilerop.CachingCompiler method)</code> , 296 |
| <code>__init__(IPython.config.loader.ArgumentParser method)</code> , 277 | <code>__init__(IPython.core.completer.Bunch method)</code> , 298 |
| <code>__init__(IPython.config.loader.CommandLineConfigLoader method)</code> , 278 | <code>__init__(IPython.core.completer.Completer method)</code> , 298 |
| <code>__init__(IPython.config.loader.Config method)</code> , 279 | <code>__init__(IPython.core.completer.CompletionSplitter method)</code> , 299 |
| <code>__init__(IPython.config.loader.ConfigError method)</code> , 280 | <code>__init__(IPython.core.completer.IPCompleter method)</code> , 300 |
| <code>__init__(IPython.config.loader.ConfigLoader method)</code> , 280 | <code>__init__(IPython.core.crashhandler.CrashHandler method)</code> , 304 |
| <code>__init__(IPython.config.loader.ConfigLoaderError method)</code> , 281 | <code>__init__(IPython.core.debugger.Pdb method)</code> , 306 |
| <code>__init__(IPython.config.loader.FileConfigLoader method)</code> , 281 | <code>__init__(IPython.core.debugger.Tracer method)</code> , 314 |
| <code>__init__(IPython.config.loader.KeyValueConfigLoader method)</code> , 282 | <code>__init__(IPython.core.display_trap.DisplayTrap method)</code> , 317 |
| | <code>__init__(IPython.core.displayhook.DisplayHook method)</code> , 319 |

`__init__(IPython.core.displaypub.DisplayPublisher
 method), 322`

`__init__(IPython.core.error.IPythonCoreError
 method), 326`

`__init__(IPython.core.error.TryNext method), 326`

`__init__(IPython.core.error.UsageError method),
 327`

`__init__(IPython.core.extensions.ExtensionManager
 method), 328`

`__init__(IPython.core.formatters.BaseFormatter
 method), 331`

`__init__(IPython.core.formatters.DisplayFormatter
 method), 334`

`__init__(IPython.core.formatters.FormatterABC
 method), 336`

`__init__(IPython.core.formatters.HTMLFormatter
 method), 337`

`__init__(IPython.core.formatters.JSONFormatter
 method), 340`

`__init__(IPython.core.formatters.JavascriptFormatter
 method), 343`

`__init__(IPython.core.formatters.LatexFormatter
 method), 345`

`__init__(IPython.core.formatters.PNGFormatter
 method), 348`

`__init__(IPython.core.formatters.PlainTextFormatter
 method), 351`

`__init__(IPython.core.formatters.SVGFormatter
 method), 355`

`__init__(IPython.core.history.HistoryManager
 method), 359`

`__init__(IPython.core.history.HistorySavingThread
 method), 363`

`__init__(IPython.core.hooks.CommandChainDispatcher
 method), 367`

`__init__(IPython.core.inputsplitter.EscapedTransformer
 method), 369`

`__init__(IPython.core.inputsplitter.IPythonInputSplitter
 method), 369`

`__init__(IPython.core.inputsplitter.InputSplitter
 method), 371`

`__init__(IPython.core.inputsplitter.LineInfo
 method), 373`

`__init__(IPython.core.interactiveshell.InteractiveShell
 method), 375`

`__init__(IPython.core.interactiveshell.InteractiveShellABC
 method), 415`

`init(IPython.core.interactiveshell.ReadLineNoRecord
 method), 476`

`method), 415`

`__init__(IPython.core.interactiveshell.SeparateUnicode
 method), 416`

`__init__(IPython.core.interactiveshell.SpaceInInput
 method), 417`

`__init__(IPython.core.logger.Logger method),
 418`

`__init__(IPython.core.macro.Macro method), 419`

`__init__(IPython.core.magic.MacroToEdit
 method), 420`

`__init__(IPython.core.magic.Magic method), 420`

`__init__(IPython.core.magic_arguments.ArgDecorator
 method), 445`

`__init__(IPython.core.magic_arguments.MagicArgumentParser
 method), 446`

`__init__(IPython.core.magic_arguments.argument
 method), 447`

`__init__(IPython.core.magic_arguments.argument_group
 method), 447`

`__init__(IPython.core.magic_arguments.kwds
 method), 447`

`__init__(IPython.core.magic_arguments.magic_arguments
 method), 447`

`__init__(IPython.core.oinspect.Inspect method),
 449`

`__init__(IPython.core.payload.PayloadManager
 method), 454`

`__init__(IPython.core.plugin.Plugin method), 457`

`__init__(IPython.core.plugin.PluginManager
 method), 459`

`__init__(IPython.core.prefilter.AliasChecker
 method), 462`

`__init__(IPython.core.prefilter.AliasHandler
 method), 463`

`__init__(IPython.core.prefilter.AssignMagicTransformer
 method), 465`

`__init__(IPython.core.prefilter.AssignSystemTransformer
 method), 467`

`__init__(IPython.core.prefilter.AssignmentChecker
 method), 469`

`__init__(IPython.core.prefilter.AutoHandler
 method), 471`

`__init__(IPython.core.prefilter.AutoMagicChecker
 method), 473`

`__init__(IPython.core.prefilter.AutocallChecker
 method), 474`

`__init__(IPython.core.prefilter.EmacsChecker
 method), 476`

| | |
|--|---|
| __init__(IPython.core.prefilter.EmacsHandler method), 478 | __init__(IPython.core.prompts.BasePrompt method), 530 |
| __init__(IPython.core.prefilter.EscCharsChecker method), 480 | __init__(IPython.core.prompts.Prompt1 method), 530 |
| __init__(IPython.core.prefilter.HelpHandler method), 482 | __init__(IPython.core.prompts.Prompt2 method), 531 |
| __init__(IPython.core.prefilter.IPyAutocallChecker method), 484 | __init__(IPython.core.prompts.PromptOut method), 531 |
| __init__(IPython.core.prefilter.IPyPromptTransformer method), 485 | __init__(IPython.core.shellapp.InteractiveShellApp method), 533 |
| __init__(IPython.core.prefilter.LineInfo method), 488 | __init__(IPython.core.ultratb.AutoFormattedTB method), 537 |
| __init__(IPython.core.prefilter.MacroChecker method), 488 | __init__(IPython.core.ultratb.ColorTB method), 539 |
| __init__(IPython.core.prefilter.MacroHandler method), 490 | __init__(IPython.core.ultratb.FormattedTB method), 540 |
| __init__(IPython.core.prefilter.MagicHandler method), 492 | __init__(IPython.core.ultratb.ListTB method), 542 |
| __init__(IPython.core.prefilter.MultiLineMagicChecker method), 493 | __init__(IPython.core.ultratb.SyntaxTB method), 543 |
| __init__(IPython.core.prefilter.PrefilterChecker method), 495 | __init__(IPython.core.ultratb.TBTools method), 545 |
| __init__(IPython.core.prefilter.PrefilterError method), 497 | __init__(IPython.core.ultratb.VerboseTB method), 546 |
| __init__(IPython.core.prefilter.PrefilterHandler method), 497 | __init__(IPython.lib.backgroundjobs.BackgroundJobBase method), 548 |
| __init__(IPython.core.prefilter.PrefilterManager method), 500 | __init__(IPython.lib.backgroundjobs.BackgroundJobExpr method), 549 |
| __init__(IPython.core.prefilter.PrefilterTransformer method), 503 | __init__(IPython.lib.backgroundjobs.BackgroundJobFunc method), 550 |
| __init__(IPython.core.prefilter.PyPromptTransformer method), 505 | __init__(IPython.lib.backgroundjobs.BackgroundJobManager method), 551 |
| __init__(IPython.core.prefilter.PythonOpsChecker method), 506 | __init__(IPython.lib.demo.ClearDemo method), 557 |
| __init__(IPython.core.prefilter.ShellEscapeChecker method), 508 | __init__(IPython.lib.demo.ClearIPDemo method), 559 |
| __init__(IPython.core.prefilter.ShellEscapeHandler method), 510 | __init__(IPython.lib.demo.ClearMixin method), 560 |
| __init__(IPython.core.profileapp.ProfileApp method), 513 | __init__(IPython.lib.demo.Demo method), 561 |
| __init__(IPython.core.profileapp.ProfileCreate method), 516 | __init__(IPython.lib.demo.DemoError method), 562 |
| __init__(IPython.core.profileapp.ProfileList method), 521 | __init__(IPython.lib.demo.IPythonDemo method), 563 |
| __init__(IPython.core.profledir.ProfileDir method), 526 | __init__(IPython.lib.demo.IPythonLineDemo method), 565 |
| __init__(IPython.core.profledir.ProfileDirError method), 529 | __init__(IPython.lib.demo.LineDemo method), 566 |
| | __init__(IPython.lib.inpthook.InputHookManager |

method), 570
__init__() (IPython.lib.irunner.IPythonRunner method), 574
__init__() (IPython.lib.irunner.InteractiveRunner method), 575
__init__() (IPython.lib.irunner.PythonRunner method), 576
__init__() (IPython.lib.irunner.RunnerFactory method), 577
__init__() (IPython.lib.irunner.SAGERunner method), 577
__init__() (IPython.lib.pretty.Breakable method), 582
__init__() (IPython.lib.pretty.Group method), 582
__init__() (IPython.lib.pretty.GroupQueue method), 582
__init__() (IPython.lib.pretty.PrettyPrinter method), 582
__init__() (IPython.lib.pretty.Printable method), 583
__init__() (IPython.lib.pretty.RepresentationPrinter method), 583
__init__() (IPython.lib.pretty.Text method), 584
__init__() (IPython.parallel.apps.baseapp.BaseParallelApplication) (IPython.parallel.apps.launcherMPIExecLauncher method), 587
__init__() (IPython.parallel.apps.baseapp.PIDFileError __init__()) (IPython.parallel.apps.launcher.PBSControllerLauncher method), 593
__init__() (IPython.parallel.apps.baseapp.ParallelCrashHandler) (IPython.parallel.apps.launcher.PBSEngineSetLauncher method), 593
__init__() (IPython.parallel.apps.ipclusterapp.IPClusterApp __init__()) (IPython.parallel.apps.launcher.PBSLauncher method), 594
__init__() (IPython.parallel.apps.ipclusterapp.IPClusterEngines) (IPython.parallel.apps.launcher.ProcessStateError method), 597
__init__() (IPython.parallel.apps.ipclusterapp.IPClusterStart __init__()) (IPython.parallel.apps.launcher.SGEControllerLauncher method), 603
__init__() (IPython.parallel.apps.ipclusterapp.IPClusterStop __init__()) (IPython.parallel.apps.launcher.SGEEngineSetLauncher method), 609
__init__() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp __init__()) (IPython.parallel.apps.launcher.SGELauncher method), 615
__init__() (IPython.parallel.apps.ipengineapp.IPEngineApp __init__()) (IPython.parallel.apps.launcher.SSHControllerLauncher method), 622
__init__() (IPython.parallel.apps.ipengineapp.MPI __init__()) (IPython.parallel.apps.launcher.SSHEngineLauncher method), 627
__init__() (IPython.parallel.apps.iploggerapp.IPLLoggerApp __init__()) (IPython.parallel.apps.launcher.SSHEngineSetLauncher method), 630
__init__() (IPython.parallel.apps.launcher.BaseLauncher __init__()) (IPython.parallel.apps.launcher.SSHLauncher method), 636
__init__() (IPython.parallel.apps.launcher.BatchSystemLauncher) (IPython.parallel.apps.launcher.UnknownStatus method), 639
__init__() (IPython.parallel.apps.launcher.IPClusterLauncher method), 642
__init__() (IPython.parallel.apps.launcher.LSFControllerLauncher method), 645
__init__() (IPython.parallel.apps.launcher.LSFEngineSetLauncher method), 649
__init__() (IPython.parallel.apps.launcher.LSFLauncher method), 652
__init__() (IPython.parallel.apps.launcher.LauncherError method), 656
__init__() (IPython.parallel.apps.launcher.LocalControllerLauncher method), 656
__init__() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 659
__init__() (IPython.parallel.apps.launcher.LocalEngineSetLauncher method), 661
__init__() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 664
__init__() (IPython.parallel.apps.launcher.MPIExecControllerLauncher method), 667
__init__() (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher method), 670

__init__(IPython.parallel.apps.launcher.WindowsHPCController) (IPython.parallel.controller.dictdb.BaseDB method), 709
method), 776
__init__(IPython.parallel.apps.launcher.WindowsHPCEngineSet) (IPython.parallel.controller.dictdb.CompositeFilter method), 712
method), 778
__init__(IPython.parallel.apps.launcher.WindowsHPCInitClient) (IPython.parallel.controller.dictdb.DictDB method), 714
method), 778
__init__(IPython.parallel.apps.logwatcher.LogWatcher) (IPython.parallel.controller.heartmonitor.Heart method), 718
method), 781
__init__(IPython.parallel.apps.win32support.Forwarder) (IPython.parallel.controller.heartmonitor.HeartMonitor method), 720
method), 781
__init__(IPython.parallel.apps.winhpcjob.IPController) (IPython.parallel.controller.hub.EngineConnector method), 721
method), 784
__init__(IPython.parallel.apps.winhpcjob.IPControllerTask) (IPython.parallel.controller.hub.Hub method), 725
method), 786
__init__(IPython.parallel.apps.winhpcjob.IPEngineSet) (IPython.parallel.controller.hub.HubFactory method), 727
method), 791
__init__(IPython.parallel.apps.winhpcjob.IPEngineTask) (IPython.parallel.controller.scheduler.TaskScheduler method), 730
method), 795
__init__(IPython.parallel.apps.winhpcjob.WinHPCJob) (IPython.parallel.controller.sqlitedb.SQLiteDB method), 733
method), 800
__init__(IPython.parallel.apps.winhpcjob.WinHPCTask) (IPython.parallel.engine.engine.EngineFactory method), 736
method), 803
__init__(IPython.parallel.client.asyncresult.AsyncHubResult) (IPython.parallel.engine.kernelstarter.KernelStarter method), 740
method), 806
__init__(IPython.parallel.client.asyncresult.AsyncMapResult) (IPython.parallel.engine.streamkernel.Kernel method), 741
method), 808
__init__(IPython.parallel.client.asyncresult.AsyncResult) (IPython.parallel.error.AbortedPendingDeferredError method), 742
method), 813
__init__(IPython.parallel.client.client.Client) (IPython.parallel.error.ClientError method), 746
method), 813
__init__(IPython.parallel.client.client.Metadata) (IPython.parallel.error.CompositeError method), 751
method), 813
__init__(IPython.parallel.client.remotefunction.ParallelFunction) (IPython.parallel.error.ConnectionError method), 755
method), 814
__init__(IPython.parallel.client.remotefunction.RemoteFunction) (IPython.parallel.error.ControllerCreationError method), 755
method), 814
__init__(IPython.parallel.client.view.DirectView) (IPython.parallel.error.ControllerError method), 757
method), 814
__init__(IPython.parallel.client.view.LoadBalancedView) (IPython.parallel.error.DependencyTimeout method), 762
method), 814
__init__(IPython.parallel.client.view.View) (IPython.parallel.error.EngineCreationError method), 768
method), 815
__init__(IPython.parallel.controller.dependency.Dependency) (IPython.parallel.error.EngineError method), 773
method), 815
__init__(IPython.parallel.controller.dependency.dependency) (IPython.parallel.error.FileTimeoutError method), 774
method), 815
__init__(IPython.parallel.controller.dependency.dependency) (IPython.parallel.error.IPythonError method), 775
method), 815

`__init__()` (IPython.parallel.error.IdInUse method), `__init__()` (IPython.parallel.error.TaskTimeout method), 816
`__init__()` (IPython.parallel.error.ImpossibleDependency `__init__()` (IPython.parallel.error.TimeoutError method), 816 method), 822
`__init__()` (IPython.parallel.error.InvalidClientID method), 816 `__init__()` (IPython.parallel.error.UnmetDependency method), 822
`__init__()` (IPython.parallel.error.InvalidDeferredID method), 816 `__init__()` (IPython.parallel.error.UnpickleableException method), 822
`__init__()` (IPython.parallel.error.InvalidDependency method), 817 `__init__()` (IPython.parallel.factory.RegistrationFactory method), 823
`__init__()` (IPython.parallel.error.InvalidEngineID method), 817 `__init__()` (IPython.parallel.util.Namespace method), 826
`__init__()` (IPython.parallel.error.InvalidProperty method), 817 `__init__()` (IPython.parallel.util.ReverseDict method), 827
`__init__()` (IPython.parallel.error.KernelError method), 817 `__init__()` (IPython.testing.globalipapp.StreamProxy method), 833
`__init__()` (IPython.parallel.error.MessageSizeError method), 818 `__init__()` (IPython.testing.globalipapp.ipnsdict method), 833
`__init__()` (IPython.parallel.error.MissingBlockArgument `__init__()` (IPython.testing.globalipapp.py_file_finder method), 818 method), 834
`__init__()` (IPython.parallel.error.NoEnginesRegistered `__init__()` (IPython.testing.ipptest.IPTester method), method), 818 835
`__init__()` (IPython.parallel.error.NotAPendingResult `__init__()` (IPython.testing.ipunittest.Doc2UnitTester method), 818 method), 837
`__init__()` (IPython.parallel.error.NotDefined `__init__()` (IPython.testing.ipunittest.IPython2PythonConverter method), 819 method), 837
`__init__()` (IPython.parallel.error.PBMessageSizeError `__init__()` (IPython.testing.mkdoctests.IndentOut method), 819 method), 839
`__init__()` (IPython.parallel.error.ProtocolError `__init__()` (IPython.testing.mkdoctests.RunnerFactory method), 819 method), 839
`__init__()` (IPython.parallel.error.QueueCleared `__init__()` (IPython.testing.plugin.show_refs.C method), 819 method), 842
`__init__()` (IPython.parallel.error.RemoteError `__init__()` (IPython.testing.tools.TempFileMixin method), 820 method), 846
`__init__()` (IPython.parallel.error.ResultAlreadyRetrieved `__init__()` (IPython.utils.PyColorize.Parser method), method), 820 849
`__init__()` (IPython.parallel.error.ResultNotCompleted `__init__()` (IPython.utils.autoattr.OneTimeProperty method), 820 method), 852
`__init__()` (IPython.parallel.error.SecurityError `__init__()` (IPython.utils.autoattr.ResetMixin method), 820 method), 853
`__init__()` (IPython.parallel.error.SerializationError `__init__()` (IPython.utils.coloransi.ColorScheme method), 821 method), 855
`__init__()` (IPython.parallel.error.StopLocalExecution `__init__()` (IPython.utils.coloransi.ColorSchemeTable method), 821 method), 855
`__init__()` (IPython.parallel.error.TaskAborted `__init__()` (IPython.utils.growl.IPythonGrowlError method), 821 method), 863
`__init__()` (IPython.parallel.error.TaskRejectError `__init__()` (IPython.utils.growl.Notifier method), method), 822 863

| | | |
|---|-----|-----|
| __init__(IPython.utils.io.IOStream method), | 865 | 904 |
| __init__(IPython.utils.io.IOTerm method), | 865 | 905 |
| __init__(IPython.utils.io.NLprinter method), | 865 | 906 |
| __init__(IPython.utils.io.Tee method), | 865 | 907 |
| __init__(IPython.utils.ipstruct.Struct method), | 868 | 908 |
| __init__(IPython.utils.newserialized.SerializationError method), | 874 | 909 |
| __init__(IPython.utils.newserialized.SerializeIt method), | 874 | 910 |
| __init__(IPython.utils.newserialized.Serialized method), | 874 | 911 |
| __init__(IPython.utils.newserialized.UnSerializeIt method), | 874 | 912 |
| __init__(IPython.utils.notification.NotificationCenter method), | 876 | 913 |
| __init__(IPython.utils.notification.NotificationError method), | 877 | 914 |
| __init__(IPython.utils.path.HomeDirError method), | 877 | 915 |
| __init__(IPython.utils.pickleshare.PickleShareDB method), | 881 | 915 |
| __init__(IPython.utils.pickleshare.PickleShareLink method), | 882 | 916 |
| __init__(IPython.utils.pickleutil.CannedFunction method), | 883 | 917 |
| __init__(IPython.utils.pickleutil.CannedObject method), | 883 | 918 |
| __init__(IPython.utils.pickleutil.Reference method), | 883 | 919 |
| __init__(IPython.utils.process.FindCmdError method), | 884 | 920 |
| __init__(IPython.utils.strdispatch.StrDispatch method), | 886 | 921 |
| __init__(IPython.utils.syspathcontext.appended_to_syspath method), | 888 | 922 |
| __init__(IPython.utils.syspathcontext.prepended_to_syspath method), | 888 | 923 |
| __init__(IPython.utils.text.EvalFormatter method), | 890 | 924 |
| __init__(IPython.utils.text.LSString method), | 890 | 925 |
| __init__(IPython.utils.text.SList method), | 895 | 926 |
| __init__(IPython.utils.traitlets.Any method), | 901 | 928 |
| __init__(IPython.utils.traitlets.Bool method), | 902 | 929 |
| __init__(IPython.utils.traitlets.Bytes method), | 903 | 930 |
| __init__(IPython.utils.traitlets.CBool method), | | 931 |

__init__(IPython.utils.traits.Type method), 932
__init__(IPython.utils.traits.Unicode method), 933

A

abbrev_cwd() (in module IPython.utils.process), 884
abort() (IPython.parallel.client.asyncresult.AsyncHubResult built-in method), 740
abort() (IPython.parallel.client.asyncresult.AsyncMapResult built-in method), 741
abort() (IPython.parallel.client.asyncresult.AsyncResult method), 742
abort() (IPython.parallel.client.client.Client method), 746
abort() (IPython.parallel.client.view.DirectView method), 757
abort() (IPython.parallel.client.view.LoadBalancedView method), 763
abort() (IPython.parallel.client.view.View method), 768
abort() (IPython.parallel.engine.engine.EngineFactory method), 803
abort_queue() (IPython.parallel.engine.streamkernel.KernelObserver method), 808
abort_queues() (IPython.parallel.engine.streamkernel.KernelObserver method), 808
abort_request() (IPython.parallel.engine.streamkernel.KernelRecord method), 808
aborted (IPython.parallel.engine.streamkernel.Kernel attribute), 808
AbortedPendingDeferredError (class in IPython.parallel.error), 813
activate() (IPython.core.builtin_trap.BuiltinTrap method), 294
activate() (IPython.parallel.client.view.DirectView method), 757
activate_matplotlib() (in module IPython.lib.pylabtools), 585
add (IPython.parallel.controller.dependency.Dependency attribute), 773
add() (IPython.core.hooks.CommandChainDispatcher method), 367
add() (IPython.lib.pretty.Text method), 584
add_argument() (IPython.config.loader.ArgumentParser method), 277
add_argument() (IPython.core.magic_arguments.MagicArgumentParser method), 446
add_argument_group()

(IPython.config.loader.ArgumentParser method), 278
add_argument_group() (IPython.core.magic_arguments.MagicArgumentParser method), 446
add_article() (in module IPython.utils.traits), 934
add_builtin() (IPython.core.builtin_trap.BuiltinTrap method), 294
add_heart_failure_handler() (IPython.parallel.controller.heartmonitor.HeartMonitor method), 782
add_job() (IPython.parallel.controller.scheduler.TaskScheduler method), 795
add_mutually_exclusive_group() (IPython.config.loader.ArgumentParser method), 278
add_mutually_exclusive_group() (IPython.core.magic_arguments.MagicArgumentParser method), 446
add_new_heart_handler() (IPython.parallel.controller.heartmonitor.HeartMonitor method), 782
add_observer() (IPython.utils.notification.NotificationCenter method), 876
add_record() (IPython.parallel.controller.dictdb.DictDB method), 778
add_record() (IPython.parallel.controller.sqlite.SQLiteDB method), 800
add_s() (IPython.utils.strdispatch.StrDispatch method), 886
add_scheme() (IPython.utils.coloransi.ColorSchemeTable method), 855
add_subparsers() (IPython.config.loader.ArgumentParser method), 278
add_subparsers() (IPython.core.magic_arguments.MagicArgumentParser method), 446
add_task() (IPython.parallel.apps.winhpcjob.IPCControllerJob method), 722
add_task() (IPython.parallel.apps.winhpcjob.IPEngineSetJob method), 728
add_task() (IPython.parallel.apps.winhpcjob.WinHPCJob method), 733
add_to_parser() (IPython.core.magic_arguments.ArgDecorator method), 445
add_to_parser() (IPython.core.magic_arguments.argument method), 447

add_to_parser() (IPython.core.magic_arguments.argument_group attribute), 622
 method), 447
add_to_parser() (IPython.core.magic_arguments.kwds method), 447
add_to_parser() (IPython.core.magic_arguments.magicAliasManager attribute), 284
 method), 447
after (IPython.parallel.client.view.LoadBalancedView attribute), 763
again() (IPython.lib.demo.ClearDemo method), 557
again() (IPython.lib.demo.ClearIPDemo method), 559
again() (IPython.lib.demo.Demo method), 561
again() (IPython.lib.demo.IPythonDemo method), 563
again() (IPython.lib.demo.IPythonLineDemo method), 565
again() (IPython.lib.demo.LineDemo method), 567
alias_manager (IPython.core.interactiveshell.InteractiveShell attribute), 375
alias_matches() (IPython.core.completer.IPCCompleter method), 300
AliasChecker (class in IPython.core.prefilter), 462
AliasError (class in IPython.core.alias), 284
aliases (IPython.config.application.Application attribute), 263
aliases (IPython.core.alias.AliasManager attribute), 284
aliases (IPython.core.application.BaseIPythonApplication attribute), 287
aliases (IPython.core.profileapp.ProfileApp attribute), 513
aliases (IPython.core.profileapp.ProfileCreate attribute), 516
aliases (IPython.core.profileapp.ProfileList attribute), 521
aliases (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 587
aliases (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 594
aliases (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 597
aliases (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 603
aliases (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 609
aliases (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 615
aliases (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 622
 method), 447
 attribute), 630
 AliasHandler (class in IPython.core.prefilter), 463
 all (IPython.parallel.controller.dependency.Dependency attribute), 773
 all_belong() (in module IPython.utils.attic), 850
 all_completed (IPython.parallel.controller.hub.Hub attribute), 786
 all_completed (IPython.parallel.controller.scheduler.TaskScheduler attribute), 795
 all_completions() (IPython.core.completer.IPCCompleter method), 300
 all_done (IPython.parallel.controller.scheduler.TaskScheduler attribute), 795
 all_failed (IPython.parallel.controller.scheduler.TaskScheduler attribute), 795
 all_ids (IPython.parallel.controller.scheduler.TaskScheduler attribute), 795
 allow_new_attr() (IPython.utils.ipstruct.Struct method), 868
 Any (class in IPython.utils.traits), 901
 append (IPython.utils.text.SList attribute), 895
 appended_to_syspath (class in IPython.utils.syspathcontext), 888
 Application (class in IPython.config.application), 263
 ApplicationError (class in IPython.config.application), 267
 apply() (IPython.parallel.client.view.DirectView method), 757
 apply() (IPython.parallel.client.view.LoadBalancedView method), 763
 apply() (IPython.parallel.client.view.View method), 768
 apply_async() (IPython.parallel.client.view.DirectView method), 757
 apply_async() (IPython.parallel.client.view.LoadBalancedView method), 763
 apply_async() (IPython.parallel.client.view.View method), 768
 apply_request() (IPython.parallel.engine.streamkernel.Kernel method), 808
 apply_sync() (IPython.parallel.client.view.DirectView method), 757
 apply_sync() (IPython.parallel.client.view.LoadBalancedView method), 763

apply_sync() (IPython.parallel.client.view.View method), 768
apply_wrapper() (in module IPython.testing.decorators), 830
arg_err() (IPython.core.interactiveshell.InteractiveShell method), 375
arg_err() (IPython.core.magic.Magic method), 420
arg_split() (in module IPython.utils.process), 884
arg_str (IPython.parallel.apps.launcher.BaseLauncher attribute), 636
arg_str (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 639
arg_str (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 642
arg_str (IPython.parallel.apps.launcher.LocalController attribute), 656
arg_str (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 659
arg_str (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 661
arg_str (IPython.parallel.apps.launcher.LocalProcessLauncher attribute), 664
arg_str (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 645
arg_str (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 649
arg_str (IPython.parallel.apps.launcher.LSFLauncher attribute), 652
arg_str (IPython.parallel.apps.launcher.MPIExecController attribute), 667
arg_str (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 670
arg_str (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 673
arg_str (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 676
arg_str (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 679
arg_str (IPython.parallel.apps.launcher.PBSLauncher attribute), 683
arg_str (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 686
arg_str (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 690
arg_str (IPython.parallel.apps.launcher.SGELauncher attribute), 693
arg_str (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 697
arg_str (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 700
arg_str (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 703
arg_str (IPython.parallel.apps.launcher.SSHLauncher attribute), 705
arg_str (IPython.parallel.apps.launcher.WindowsHPCCControllerLauncher attribute), 709
arg_str (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher attribute), 712
arg_str (IPython.parallel.apps.launcher.WindowsHPCLauncher attribute), 715
ArgDecorator (class in IPython.core.magic_arguments), 445
ArgParserConfigLoader (class in IPython.config.loader), 276
Argument (IPython.config.application.ApplicationError attribute), 267
Argument (IPython.config.configurable.ConfigurableError attribute), 270
Argument (IPython.config.configurable.MultipleInstanceError attribute), 272
Argument (IPython.config.loader.ArgumentParser attribute), 277
Argument (IPython.config.loader.ConfigError attribute), 280
args (IPython.config.loader.ConfigLoaderError attribute), 281
args (IPython.core.alias.AliasError attribute), 284
args (IPython.core.alias.InvalidAliasError attribute), 285
args (IPython.core.error.IPythonCoreError attribute), 326
args (IPython.core.error.TryNext attribute), 326
args (IPython.core.error.UsageError attribute), 327
args (IPython.core.interactiveshell.SpaceInInput attribute), 417
args (IPython.core.magic.MacroToEdit attribute), 420
args (IPython.core.prefilter.PrefilterError attribute), 497
args (IPython.core.profiledir.ProfileDirError attribute), 529
args (IPython.lib.demo.DemoError attribute), 562
args (IPython.parallel.apps.baseapp.PIDFileError attribute), 593
args (IPython.parallel.apps.launcher.BaseLauncher attribute), 636

args (IPython.parallel.apps.launcher.BatchSystemLauncher args (IPython.parallel.apps.launcher.UnknownStatus attribute), 639
attribute), 708
args (IPython.parallel.apps.launcher.IPClusterLauncher args (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher attribute), 642
attribute), 709
args (IPython.parallel.apps.launcher.LauncherError args (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher attribute), 656
attribute), 712
args (IPython.parallel.apps.launcher.LocalControllerLauncher args (IPython.parallel.apps.launcher.WindowsHPCLauncher attribute), 656
attribute), 715
args (IPython.parallel.apps.launcher.LocalEngineLauncher args (IPython.parallel.error.AbortedPendingDeferredError attribute), 659
attribute), 813
args (IPython.parallel.apps.launcher.LocalEngineSetLauncher args (IPython.parallel.error.ClientError attribute), 662
attribute), 813
args (IPython.parallel.apps.launcher.LocalProcessLauncher args (IPython.parallel.error.CompositeError attribute), 664
attribute), 813
args (IPython.parallel.apps.launcher.LSFControllerLauncher args (IPython.parallel.error.ConnectionError attribute), 645
attribute), 814
args (IPython.parallel.apps.launcher.LSFEngineSetLauncher args (IPython.parallel.error.ControllerCreationError attribute), 649
attribute), 814
args (IPython.parallel.apps.launcher.LSFLauncher args (IPython.parallel.error.ControllerError attribute), 652
attribute), 814
args (IPython.parallel.apps.launcher.MPIExecControllerLauncher args (IPython.parallel.error.DependencyTimeout attribute), 667
attribute), 814
args (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher args (IPython.parallel.error.EngineCreationError attribute), 670
attribute), 815
args (IPython.parallel.apps.launcher.MPIExecLauncher args (IPython.parallel.error.EngineError attribute), 673
attribute), 815
args (IPython.parallel.apps.launcher.PBSControllerLauncher args (IPython.parallel.error.FileTimeoutError attribute), 676
attribute), 815
args (IPython.parallel.apps.launcher.PBSEngineSetLauncher args (IPython.parallel.error.IdInUse attribute), 679
attribute), 816
args (IPython.parallel.apps.launcher.PBSLauncher args (IPython.parallel.error.ImpossibleDependency attribute), 683
attribute), 816
args (IPython.parallel.apps.launcher.ProcessStateError args (IPython.parallel.error.InvalidClientID attribute), 686
attribute), 816
args (IPython.parallel.apps.launcher.SGEControllerLauncher args (IPython.parallel.error.InvalidDeferredID attribute), 686
attribute), 816
args (IPython.parallel.apps.launcher.SGEEngineSetLauncher args (IPython.parallel.error.InvalidDependency attribute), 690
attribute), 817
args (IPython.parallel.apps.launcher.SGELauncher args (IPython.parallel.error.InvalidEngineID attribute), 693
attribute), 817
args (IPython.parallel.apps.launcher.SSHControllerLauncher args (IPython.parallel.error.InvalidProperty attribute), 697
attribute), 817
args (IPython.parallel.apps.launcher.SSHEngineLauncher args (IPython.parallel.error.IPythonError attribute), 700
attribute), 815
args (IPython.parallel.apps.launcher.SSHEngineSetLauncher args (IPython.parallel.error.KernelError attribute), 703
attribute), 817
args (IPython.parallel.apps.launcher.SSHLauncher args (IPython.parallel.error.MessageSizeError attribute), 705
attribute), 818
args (IPython.parallel.error.MissingBlockArgument

attribute), 818
args (IPython.parallel.error.NoEnginesRegistered attribute), 818
args (IPython.parallel.error.NotAPendingResult attribute), 818
args (IPython.parallel.error.NotDefined attribute), 819
args (IPython.parallel.error.PBMessageSizeError attribute), 819
args (IPython.parallel.error.ProtocolError attribute), 819
args (IPython.parallel.error.QueueCleared attribute), 819
args (IPython.parallel.error.RemoteError attribute), 820
args (IPython.parallel.error.ResultAlreadyRetrieved attribute), 820
args (IPython.parallel.error.ResultNotCompleted attribute), 820
args (IPython.parallel.error.SecurityError attribute), 820
args (IPython.parallel.error.SerializationError attribute), 821
args (IPython.parallel.error.StopLocalExecution attribute), 821
args (IPython.parallel.error.TaskAborted attribute), 821
args (IPython.parallel.error.TaskRejectError attribute), 822
args (IPython.parallel.error.TaskTimeout attribute), 822
args (IPython.parallel.error.TimeoutError attribute), 822
args (IPython.parallel.error.UnmetDependency attribute), 822
args (IPython.parallel.error.UnpickleableException attribute), 823
args (IPython.utils.growl.IPythonGrowlError attribute), 863
args (IPython.utils.newserialized.SerializationError attribute), 874
args (IPython.utils.notification.NotificationError attribute), 877
args (IPython.utils.path.HomeDirError attribute), 877
args (IPython.utils.process.FindCmdError attribute), 884
args (IPython.utils.traits.TraitError attribute), 929
argument (class in IPython.core.magic_arguments), 447
argument_group (class in IPython.core.magic_arguments), 447
ArgumentError (class in IPython.config.loader), 277
ArgumentParser (class in IPython.config.loader), 277
as_dict() (IPython.parallel.controller.dependency.Dependency method), 773
as_element() (IPython.parallel.apps.winhpcjob.IPControllerJob method), 722
as_element() (IPython.parallel.apps.winhpcjob.IPControllerTask method), 725
as_element() (IPython.parallel.apps.winhpcjob.IPEngineSetJob method), 728
as_element() (IPython.parallel.apps.winhpcjob.IPEngineTask method), 730
as_element() (IPython.parallel.apps.winhpcjob.WinHPCJob method), 733
as_element() (IPython.parallel.apps.winhpcjob.WinHPCTask method), 737
as_str() (in module IPython.parallel.apps.winhpcjob), 739
as_unittest() (in module IPython.testing.decorators), 830
asbytes() (in module IPython.parallel.util), 828
ask_yes_no() (in module IPython.utils.io), 866
ask_yes_no() (IPython.core.interactiveshell.InteractiveShell method), 376
AssignMagicTransformer (class in IPython.core.prefilter), 465
AssignmentChecker (class in IPython.core.prefilter), 469
AssignSystemTransformer (class in IPython.core.prefilter), 467
AsyncResult (built-in class), 169
AsyncResult (class in IPython.parallel.client.asyncresult), 742
atexit_operations() (IPython.core.interactiveshell.InteractiveShell method), 376
attr_matches() (IPython.core.completer.Completer method), 299
attr_matches() (IPython.core.completer.IPCompleter method), 300

audit_timeouts() (IPython.parallel.controller.scheduler.TaskSchedulerTB (class in IPython.core.ultratb), method), 795
 auditor (IPython.parallel.controller.scheduler.TaskSchedulerHandler (class in IPython.core.prefilter), 471 attribute), 795
 auto_attr() (in module IPython.utils.autoattr), 853
 auto_calculate_max (IPython.parallel.apps.winhpcjob.IPCConfig (IPython.core.interactiveshell.InteractiveShell attribute), 722
 auto_calculate_max (IPython.parallel.apps.winhpcjob.IREngineMessageChecker (class in IPython.core.prefilter), attribute), 728
 auto_calculate_max (IPython.parallel.apps.winhpcjob.IREngineMessageChecker (class in IPython.core.prefilter), attribute), 473
 auto_calculate_max (IPython.parallel.apps.winhpcjob.WinHPCJob B
 attribute), 733
 auto_calculate_min (IPython.parallel.apps.winhpcjob.IPCConfig (IPython.lib.demo.ClearDemo method), 557 attribute), 722
 back() (IPython.lib.demo.ClearIDemo method),
 auto_calculate_min (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 728
 back() (IPython.lib.demo.Demo method), 561
 auto_calculate_min (IPython.parallel.apps.winhpcjob.WinHPCJob (IPython.lib.demo.IPythonDemo method), attribute), 734
 563
 auto_create (IPython.core.application.BaseIPythonApplication back() (IPython.lib.demo.IPythonLineDemo attribute), 287
 method), 565
 auto_create (IPython.core.profileapp.ProfileCreate back() (IPython.lib.demo.LineDemo method), 567 attribute), 516
 BackgroundJobBase (class in IPython.lib.backgroundjobs), 548
 auto_create (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 587
 BackgroundJobExpr (class in IPython.lib.backgroundjobs), 549
 auto_create (IPython.parallel.apps.ipclusterapp.IPClusterEngineIPython.lib.backgroundjobs), 549 attribute), 597
 BackgroundJobFunc (class in IPython.lib.backgroundjobs), 550
 auto_create (IPython.parallel.apps.ipclusterapp.IPClusterStart IPython.lib.backgroundjobs), 550 attribute), 603
 BackgroundJobManager (class in IPython.lib.backgroundjobs), 551
 auto_create (IPython.parallel.apps.ipclusterapp.IPClusterStop IPython.lib.backgroundjobs), 609
 BaseDB (class in IPython.parallel.controller.dictdb),
 auto_create (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp A76
 attribute), 615
 BaseFormatter (class in IPython.core.formatters),
 auto_create (IPython.parallel.apps.ipengineapp.IPEngineApp 331
 attribute), 622
 BaseIPythonApplication (class in IPython.core.formatters), 622
 auto_create (IPython.parallel.apps.iploggerapp.IPLLoggerApp IPython.core.application), 287
 attribute), 630
 BaseLauncher (class in IPython.core.application), 287
 auto_rewrite() (IPython.core.prompts.Prompt1 IPython.parallel.apps.launcher), 636
 method), 530
 BaseParallelApplication (class in IPython.parallel.apps.launcher), 636
 auto_rewrite_input() IPython.parallel.apps.baseapp), 587
 (IPython.core.interactiveshell.InteractiveShellBasePrompt (class in IPython.core.prompts), 530
 method), 376
 batch_file (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 639
 auto_status (IPython.core.interactiveshell.InteractiveShell attribute), 376
 batch_file (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 645
 auto_status (IPython.core.magic.Magic attribute), 421
 batch_file (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 649
 autocall (IPython.core.interactiveshell.InteractiveShell attribute), 376
 batch_file (IPython.parallel.apps.launcher.LSFLauncher attribute), 652
 AutocallChecker (class in IPython.core.prefilter), 474
 attribute), 652

batch_file (IPython.parallel.apps.launcher.PBSCControllerBatchTemplate (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 676 attribute), 690
batch_file (IPython.parallel.apps.launcher.PBSEngineSetBatchTemplate (IPython.parallel.apps.launcher.SGELauncher attribute), 679 attribute), 694
batch_file (IPython.parallel.apps.launcher.PBSLauncherBatchTemplate (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 683 attribute), 639
batch_file (IPython.parallel.apps.launcher.SGEControllerBatchTemplate (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 687 attribute), 645
batch_file (IPython.parallel.apps.launcher.SGEEngineSetBatchTemplate (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 690 attribute), 649
batch_file (IPython.parallel.apps.launcher.SGELauncherBatchTemplate (IPython.parallel.apps.launcher.LSFLauncher attribute), 693 attribute), 652
batch_file_name (IPython.parallel.apps.launcher.BatchSystemTemplate (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 639 attribute), 676
batch_file_name (IPython.parallel.apps.launcher.LSFCBatchTemplate (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 645 attribute), 679
batch_file_name (IPython.parallel.apps.launcher.LSFEBatchTemplate (IPython.parallel.apps.launcher.PBSLauncher attribute), 649 attribute), 683
batch_file_name (IPython.parallel.apps.launcher.LSFLBatchTemplate (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 652 attribute), 687
batch_file_name (IPython.parallel.apps.launcher.PBSCBatchTemplate (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 676 attribute), 690
batch_file_name (IPython.parallel.apps.launcher.PBSEBatchTemplate (IPython.parallel.apps.launcher.SGELauncher attribute), 679 attribute), 694
batch_file_name (IPython.parallel.apps.launcher.PBSLBatchSystemLauncher (class in IPython.parallel.apps.launcher), 639 attribute), 683
batch_file_name (IPython.parallel.apps.launcher.SGEQBdBatchTemplate (IPython.core.debugger), 314 attribute), 687
batch_file_name (IPython.parallel.apps.launcher.SGEQBdQSetIPython_excepthook (IPython.core.debugger), 314 attribute), 690
batch_file_name (IPython.parallel.apps.launcher.SGELBatchTemplate (IPython.parallel.controller.HeartMonitor attribute), 693 method), 782
batch_template (IPython.parallel.apps.launcher.BatchSystemTemplate (IPython.lib.pretty.PrettyPrinter attribute), 639 method), 582
batch_template (IPython.parallel.apps.launcher.LSFCohortGroup (IPython.lib.pretty.RepresentationPrinter attribute), 645 method), 583
batch_template (IPython.parallel.apps.launcher.LSFEngineSet (in module IPython.utils.attic), 850 attribute), 649 bident (IPython.parallel.engine.EngineFactory attribute), 850
batch_template (IPython.parallel.apps.launcher.LSFLauncher attribute), 803 bident (IPython.parallel.engine.streamkernel.Kernel attribute), 652
batch_template (IPython.parallel.apps.launcher.PBSControllerAttribute), 808 Black (IPython.utils.coloransi.InputTermColors attribute), 676
batch_template (IPython.parallel.apps.launcher.PBSEngineSetAttribute), 856 Black (IPython.utils.coloransi.TermColors attribute), 679
batch_template (IPython.parallel.apps.launcher.PBSLauncher attribute), 857 blacklist (IPython.parallel.controller.scheduler.TaskScheduler attribute), 683
batch_template (IPython.parallel.apps.launcher.SGEControllerAttribute), 795 BlinkBlack (IPython.utils.coloransi.InputTermColors attribute), 687

attribute), 856
 BlinkBlack (IPython.utils.coloransi.TermColors attribute), 857
 BlinkBlue (IPython.utils.coloransi.InputTermColors attribute), 856
 BlinkBlue (IPython.utils.coloransi.TermColors attribute), 857
 BlinkCyan (IPython.utils.coloransi.InputTermColors attribute), 856
 BlinkCyan (IPython.utils.coloransi.TermColors attribute), 858
 BlinkGreen (IPython.utils.coloransi.InputTermColors attribute), 857
 BlinkGreen (IPython.utils.coloransi.TermColors attribute), 858
 BlinkLightGray (IPython.utils.coloransi.InputTermColors attribute), 857
 BlinkLightGray (IPython.utils.coloransi.TermColors attribute), 858
 BlinkPurple (IPython.utils.coloransi.InputTermColors attribute), 857
 BlinkPurple (IPython.utils.coloransi.TermColors attribute), 858
 BlinkRed (IPython.utils.coloransi.InputTermColors attribute), 857
 BlinkRed (IPython.utils.coloransi.TermColors attribute), 858
 BlinkYellow (IPython.utils.coloransi.InputTermColors attribute), 857
 BlinkYellow (IPython.utils.coloransi.TermColors attribute), 858
 block (IPython.parallel.client.Client attribute), 746
 block (IPython.parallel.client.remotefunction.ParallelFunction attribute), 755
 block (IPython.parallel.client.remotefunction.RemoteFunction attribute), 755
 block (IPython.parallel.client.view.DirectView attribute), 757
 block (IPython.parallel.client.view.LoadBalancedView attribute), 763
 block (IPython.parallel.client.view.View attribute), 768
 Blue (IPython.utils.coloransi.InputTermColors attribute), 857
 Blue (IPython.utils.coloransi.TermColors attribute), 858
 Bool (class in IPython.utils.traits), 902
 boolean_flag() (in module IPython.config.application), 267
 bp_commands() (IPython.core.debugger.Pdb method), 306
 break_anywhere() (IPython.core.debugger.Pdb method), 306
 break_here() (IPython.core.debugger.Pdb method), 306
 Breakable (class in IPython.lib.pretty), 582
 breakable() (IPython.lib.pretty.PrettyPrinter method), 583
 breakable() (IPython.lib.pretty.RepresentationPrinter method), 584
 Brown (IPython.utils.coloransi.InputTermColors attribute), 857
 Brown (IPython.utils.coloransi.TermColors attribute), 858
 build_launcher() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 597
 build_launcher() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 603
 builtin_profile_dir (IPython.core.application.BaseIPythonApplication attribute), 287
 builtin_profile_dir (IPython.core.profileapp.ProfileCreate attribute), 517
 builtin_profile_dir (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 587
 builtin_profile_dir (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 598
 builtin_profile_dir (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 603
 builtin_profile_dir (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 609
 builtin_profile_dir (IPython.parallel.apps.ipcontrollerapp.IPController attribute), 615
 builtin_profile_dir (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 622
 builtin_profile_dir (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 630
 builtin_trap (IPython.core.interactiveshell.InteractiveShell attribute), 376
 BuiltinTrap (class in IPython.core.builtin_trap), 294
 Bunch (class in IPython.core.completer), 298
 Bunch (class in IPython.core.interactiveshell), 375
 Bunch (class in IPython.core.magic), 420
 by_ident (IPython.parallel.controller.hub.Hub attribute), 787
 Bytes (class in IPython.utils.traits), 903

C

C (class in IPython.testing.plugin.show_refs), [842](#)
cache() (IPython.core.compilerop.CachingCompiler method), [296](#)
cache_main_mod() (IPython.core.interactiveshell.InteractiveShell method), [376](#)
cache_size (IPython.core.interactiveshell.InteractiveShell attribute), [377](#)
CachingCompiler (class in IPython.core.compilerop), [296](#)
call() (IPython.lib.backgroundjobs.BackgroundJobExp method), [549](#)
call() (IPython.lib.backgroundjobs.BackgroundJobFun method), [550](#)
call_alias() (IPython.core.alias.AliasManager method), [284](#)
call_args (IPython.testing.iptest.IPTester attribute), [835](#)
call_pdb (IPython.core.interactiveshell.InteractiveShell attribute), [377](#)
call_tip() (in module IPython.core.oinspect), [451](#)
can() (in module IPython.utils.pickleutil), [883](#)
canDict() (in module IPython.utils.pickleutil), [883](#)
CannedFunction (class in IPython.utils.pickleutil), [883](#)
CannedObject (class in IPython.utils.pickleutil), [883](#)
canonic() (IPython.core.debugger.Pdb method), [306](#)
canSequence() (in module IPython.utils.pickleutil), [883](#)
capitalize (IPython.utils.text.LSString attribute), [890](#)
CaselessStrEnum (class in IPython.utils.traits), [911](#)
CBool (class in IPython.utils.traits), [904](#)
CBytes (class in IPython.utils.traits), [905](#)
CCComplex (class in IPython.utils.traits), [906](#)
cd_completer() (in module IPython.core.completerlib), [303](#)
center (IPython.utils.text.LSString attribute), [890](#)
CFloat (class in IPython.utils.traits), [907](#)
check() (IPython.core.prefilter.AliasChecker method), [462](#)
check() (IPython.core.prefilter.AssignmentChecker method), [469](#)
check() (IPython.core.prefilter.AutocallChecker method), [475](#)
check() (IPython.core.prefilter.AutoMagicChecker method), [473](#)
check() (IPython.core.prefilter.EmacsChecker method), [476](#)
check() (IPython.core.prefilter.EscCharsChecker method), [480](#)
check() (IPython.core.prefilter.IPyAutocallChecker method), [484](#)
check() (IPython.core.prefilter.MacroChecker method), [488](#)
in check() (IPython.core.prefilter.MultiLineMagicChecker method), [493](#)
check() (IPython.core.prefilter.PrefilterChecker method), [495](#)
check() (IPython.core.prefilter.PythonOpsChecker method), [507](#)
check() (IPython.core.prefilter.ShellEscapeChecker method), [508](#)
check() (IPython.parallel.controller.dependency.Dependency method), [773](#)
check_aborted() (IPython.parallel.engine.streamkernel.Kernel method), [808](#)
check_cache() (IPython.core.compilerop.CachingCompiler method), [297](#)
check_dependencies() (IPython.parallel.engine.streamkernel.Kernel method), [808](#)
check_dirs() (IPython.core.profiledir.ProfileDir method), [526](#)
check_for_old_config() (in module IPython.utils.path), [878](#)
check_for_underscore() (IPython.core.displayhook.DisplayHook method), [319](#)
check_load() (IPython.parallel.controller.hub.Hub method), [787](#)
check_log_dir() (IPython.core.profiledir.ProfileDir method), [526](#)
check_pairs() (in module IPython.testing.tools), [846](#)
check_pid() (IPython.parallel.apps.baseapp.BaseParallelApplication method), [587](#)
check_pid() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), [598](#)
check_pid() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), [603](#)
check_pid() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), [609](#)
check_pid() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp method), [615](#)
check_pid() (IPython.parallel.apps.ipengineapp.IPEngineApp

method), 622
`check_pid()` (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 630
`check_pid_dir()` (IPython.core.profiledir.ProfileDir method), 526
`check_ready()` (in module IPython.parallel.client.asyncresult), 743
`check_security_dir()` (IPython.core.profiledir.ProfileDir method), 526
`check_unused_args()` (IPython.utils.text.EvalFormatter method), 890
`checkers` (IPython.core.prefilter.PrefilterManager attribute), 500
`checkline()` (IPython.core.debugger.Pdb method), 306
`children` (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 615
`chop()` (in module IPython.utils.data), 859
`chunksize` (IPython.parallel.client.remotefunction.ParallelFunctioning attribute), 755
`CInt` (class in IPython.utils.traits), 908
`class_config_section()`
 (IPython.config.application.Application class method), 263
`class_config_section()`
 (IPython.config.configurable.Configurable class method), 269
`class_config_section()`
 (IPython.config.configurable.LoggingConfigurable class method), 271
`class_config_section()`
 (IPython.config.configurable.SingletonConfigurable class method), 273
`class_config_section()`
 (IPython.core.alias.AliasManager method), 284
`class_config_section()`
 (IPython.core.application.BaseIPythonApplication class method), 287
`class_config_section()`
 (IPython.core.builtin_trap.BuiltinTrap class method), 294
`class_config_section()`
 (IPython.core.display_trap.DisplayTrap class method), 317
`class_config_section()`

(IPython.core.displayhook.DisplayHook class method), 319
`class_config_section()`
 (IPython.core.displaypub.DisplayPublisher class method), 322
`class_config_section()`
 (IPython.core.extensions.ExtensionManager class method), 328
`class_config_section()`
 (IPython.core.formatters.BaseFormatter class method), 331
`class_config_section()`
 (IPython.core.formatters.DisplayFormatter class method), 334
`class_config_section()`
 (IPython.core.formatters.HTMLFormatter class method), 337
`class_config_section()`
 (IPython.core.formatters.JavascriptFormatter class method), 343
`class_config_section()`
 (IPython.core.formatters.JSONFormatter class method), 340
`class_config_section()`
 (IPython.core.formatters.LatexFormatter class method), 346
`class_config_section()`
 (IPython.core.formatters.PlainTextFormatter class method), 352
`class_config_section()`
 (IPython.core.formatters.PNGFormatter class method), 349
`class_config_section()`
 (IPython.core.formatters.SVGFormatter class method), 355
`class_config_section()`
 (IPython.core.history.HistoryManager class method), 359
`class_config_section()`
 (IPython.core.interactiveshell.InteractiveShell class method), 377
`class_config_section()`
 (IPython.core.payload.PayloadManager class method), 454
`class_config_section()`
 (IPython.core.plugin.Plugin class method), 457
`class_config_section()`
 (IPython.core.plugin.PluginManager

class method), 459

class_config_section()
 (IPython.core.prefilter.AliasChecker
 class method), 462

class_config_section()
 (IPython.core.prefilter.AliasHandler
 class method), 463

class_config_section()
 (IPython.core.prefilter.AssignMagicTransformer
 class method), 465

class_config_section()
 (IPython.core.prefilter.AssignmentChecker
 class method), 469

class_config_section()
 (IPython.core.prefilter.AssignSystemTransformer
 class method), 467

class_config_section()
 (IPython.core.prefilter.AutocallChecker
 class method), 475

class_config_section()
 (IPython.core.prefilter.AutoHandler
 class method), 471

class_config_section()
 (IPython.core.prefilter.AutoMagicChecker
 class method), 473

class_config_section()
 (IPython.core.prefilter.EmacsChecker
 class method), 476

class_config_section()
 (IPython.core.prefilter.EmacsHandler
 class method), 478

class_config_section()
 (IPython.core.prefilter.EscCharsChecker
 class method), 480

class_config_section()
 (IPython.core.prefilter.HelpHandler
 class method), 482

class_config_section()
 (IPython.core.prefilter.IPyAutocallChecker
 class method), 484

class_config_section()
 (IPython.core.prefilter.IPyPromptTransformer
 class method), 486

class_config_section()
 (IPython.core.prefilter.MacroChecker
 class method), 488

class_config_section()
 (IPython.core.prefilter.MacroHandler
 class method), 490

class_config_section()
 (IPython.core.prefilter.MagicHandler
 class method), 492

class_config_section()
 (IPython.core.prefilter.MultiLineMagicChecker
 class method), 493

class_config_section()
 (IPython.core.prefilter.PrefilterChecker
 class method), 495

class_config_section()
 (IPython.core.prefilter.PrefilterHandler
 class method), 497

class_config_section()
 (IPython.core.prefilter.PrefilterManager
 class method), 500

class_config_section()
 (IPython.core.prefilter.PrefilterTransformer
 class method), 503

class_config_section()
 (IPython.core.prefilter.PyPromptTransformer
 class method), 505

class_config_section()
 (IPython.core.prefilter.PythonOpsChecker
 class method), 507

class_config_section()
 (IPython.core.prefilter.ShellEscapeChecker
 class method), 508

class_config_section()
 (IPython.core.prefilter.ShellEscapeHandler
 class method), 510

class_config_section()
 (IPython.core.profileapp.ProfileApp
 class method), 513

class_config_section()
 (IPython.core.profileapp.ProfileCreate
 class method), 517

class_config_section()
 (IPython.core.profileapp.ProfileList
 class method), 521

class_config_section()
 (IPython.core.profiledir.ProfileDir
 class method), 526

class_config_section()
 (IPython.core.shellapp.InteractiveShellApp
 class method), 533

class_config_section()
 (IPython.parallel.apps.baseapp.BaseParallelApplication
 class method), 534

| | |
|--|---|
| class method), 587 | class method), 646 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 594 | (IPython.parallel.apps.launcher.LSFEngineSetLauncher class method), 649 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipclusterapp.IPClusterEngines class method), 598 | (IPython.parallel.apps.launcher.LSFLauncher class method), 652 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipclusterapp.IPClusterStart class method), 603 | (IPython.parallel.apps.launcher.MPIExecControllerLauncher class method), 667 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipclusterapp.IPClusterStop class method), 609 | (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher class method), 670 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipcontrollerapp.IPControllerApp class method), 615 | (IPython.parallel.apps.launcher.MPIExecLauncher class method), 673 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipengineapp.IPEngineApp class method), 622 | (IPython.parallel.apps.launcher.PBSControllerLauncher class method), 676 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.ipengineapp.MPI class method), 628 | (IPython.parallel.apps.launcher.PBSEngineSetLauncher class method), 680 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.iploggerapp.IPLLoggerApp class method), 630 | (IPython.parallel.apps.launcher.PBSLauncher class method), 683 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.BaseLauncher class method), 636 | (IPython.parallel.apps.launcher.SGEControllerLauncher class method), 687 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.BatchSystemLauncher class method), 639 | (IPython.parallel.apps.launcher.SGEEngineSetLauncher class method), 690 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.IPClusterLauncher class method), 642 | (IPython.parallel.apps.launcher.SGELauncher class method), 694 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.LocalControllerLaunch class method), 656 | (IPython.parallel.apps.launcher.SSHControllerLauncher class method), 697 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.LocalEngineLauncher class method), 659 | (IPython.parallel.apps.launcher.SSHEngineLauncher class method), 700 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.LocalEngineSetLaunch class method), 662 | (IPython.parallel.apps.launcher.SSHEngineSetLauncher class method), 703 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.LocalProcessLauncher class method), 664 | (IPython.parallel.apps.launcher.SSHLauncher class method), 706 |
| class_config_section() | class_config_section() |
| (IPython.parallel.apps.launcher.LSFControllerLaunch class method), 667 | (IPython.parallel.apps.launcher.WindowsHPCCControllerLauncher class method), 707 |

class method), 709
class_config_section()
 (IPython.parallel.apps.launcher.WindowsHPCEngineSection, 712
 class method), 712
class_config_section()
 (IPython.parallel.apps.launcher.WindowsHPCLaunchSection, 715
 class method), 715
class_config_section()
 (IPython.parallel.apps.logwatcher.LogWatcher
 class method), 718
class_config_section()
 (IPython.parallel.apps.winhpcjob.IPControllerJob
 class method), 722
class_config_section()
 (IPython.parallel.apps.winhpcjob.IPControllerTask, 725
 class method), 725
class_config_section()
 (IPython.parallel.apps.winhpcjob.IPEngineSetJob
 class method), 728
class_config_section()
 (IPython.parallel.apps.winhpcjob.IPEngineTask, 730
 class method), 730
class_config_section()
 (IPython.parallel.apps.winhpcjob.WinHPCJob
 class method), 734
class_config_section()
 (IPython.parallel.apps.winhpcjob.WinHPCTask, 737
 class method), 737
class_config_section()
 (IPython.parallel.controller.dictdb.BaseDB
 class method), 776
class_config_section()
 (IPython.parallel.controller.dictdb.DictDB
 class method), 779
class_config_section()
 (IPython.parallel.controller.heartmonitor.HeartMonitor
 class method), 782
class_config_section()
 (IPython.parallel.controller.hub.Hub
 class method), 787
class_config_section()
 (IPython.parallel.controller.hub.HubFactory
 class method), 791
class_config_section()
 (IPython.parallel.controller.scheduler.TaskScheduler, 795
 class method), 795
class_config_section()
 (IPython.parallel.controller.sqlitedb.SQLiteDB
 class method), 349
 class method), 800
 class_config_section()
 (IPython.parallel.engine.EngineFactory
 class method), 803
 class_config_section()
 (IPython.parallel.engine.streamkernel.Kernel
 class method), 808
 class_config_section()
 (IPython.parallel.factory.RegistrationFactory
 class method), 823
 class_get_help(), (IPython.config.application.Application
 class method), 264
 class_get_help(), (IPython.config.configurable.Configurable
 class method), 269
 task_get_help(), (IPython.config.configurable.LoggingConfigurable
 class method), 271
 class_get_help(), (IPython.config.configurable.SingletonConfigurable
 class method), 273
 class_get_help(), (IPython.core.alias.AliasManager
 class method), 284
 class_get_help(), (IPython.core.application.BaseIPythonApplication
 class method), 287
 class_get_help(), (IPython.core.builtin_trap.BuiltinTrap
 class method), 294
 class_get_help(), (IPython.core.display_trap.DisplayTrap
 class method), 317
 task_get_help(), (IPython.core.displayhook.DisplayHook
 class method), 319
 class_get_help(), (IPython.core.displaypub.DisplayPublisher
 class method), 323
 class_get_help(), (IPython.core.extensions.ExtensionManager
 class method), 328
 class_get_help(), (IPython.core.formatters.BaseFormatter
 class method), 331
 class_get_help(), (IPython.core.formatters.DisplayFormatter
 class method), 334
 class_get_help(), (IPython.core.formatters.HTMLFormatter
 class method), 337
 class_get_help(), (IPython.core.formatters.JavascriptFormatter
 class method), 343
 class_get_help(), (IPython.core.formatters.JSONFormatter
 class method), 340
 class_get_help(), (IPython.core.formatters.LatexFormatter
 class method), 346
 class_get_help(), (IPython.core.formatters.PlainTextFormatter
 class method), 352
 class_get_help(), (IPython.core.formatters.PNGFormatter
 class method), 349

class_get_help() (IPython.core.formatters.SVGFormatter class method), 355
class_get_help() (IPython.core.history.HistoryManager class method), 359
class_get_help() (IPython.core.interactiveshell.InteractiveShell class method), 377
class_get_help() (IPython.core.payload.PayloadManager class method), 454
class_get_help() (IPython.core.plugin.Plugin class method), 457
class_get_help() (IPython.core.plugin.PluginManager class method), 459
class_get_help() (IPython.core.prefilter.AliasChecker class method), 462
class_get_help() (IPython.core.prefilter.AliasHandler class method), 464
class_get_help() (IPython.core.prefilter.AssignMagicTransformer class method), 465
class_get_help() (IPython.core.prefilter.AssignmentChecker class method), 469
class_get_help() (IPython.core.prefilter.AssignSystemTransformer class method), 467
class_get_help() (IPython.core.prefilter.AutocallChecker class method), 475
class_get_help() (IPython.core.prefilter.AutoHandler class method), 471
class_get_help() (IPython.core.prefilter.AutoMagicChecker class method), 473
class_get_help() (IPython.core.prefilter.EmacsChecker class method), 476
class_get_help() (IPython.core.prefilter.EmacsHandler class method), 478
class_get_help() (IPython.core.prefilter.EscCharsChecker class method), 480
class_get_help() (IPython.core.prefilter.HelpHandler class method), 482
class_get_help() (IPython.core.prefilter.IPyAutocallChecker class method), 484
class_get_help() (IPython.core.prefilter.IPyPromptTransformer class method), 486
class_get_help() (IPython.core.prefilter.MacroChecker class method), 488
class_get_help() (IPython.core.prefilter.MacroHandler class method), 490
class_get_help() (IPython.core.prefilter.MagicHandler class method), 492
class_get_help() (IPython.core.prefilter.MultiLineMagicChecker class method), 494
class_get_help() (IPython.core.prefilter.PrefilterChecker class method), 495
class_get_help() (IPython.core.prefilter.PrefilterHandler class method), 497
class_get_help() (IPython.core.prefilter.PrefilterManager class method), 500
class_get_help() (IPython.core.prefilter.PrefilterTransformer class method), 503
class_get_help() (IPython.core.prefilter.PyPromptTransformer class method), 505
class_get_help() (IPython.core.prefilter.PythonOpsChecker class method), 507
class_get_help() (IPython.core.prefilter.ShellEscapeChecker class method), 508
class_get_help() (IPython.core.prefilter.ShellEscapeHandler class method), 510
class_get_help() (IPython.core.profileapp.ProfileApp class method), 513
class_get_help() (IPython.core.profileapp.ProfileCreate class method), 517
class_get_help() (IPython.core.profileapp.ProfileList class method), 521
class_get_help() (IPython.core.profiledir.ProfileDir class method), 526
class_get_help() (IPython.core.shellapp.InteractiveShellApp class method), 533
class_get_help() (IPython.parallel.apps.baseapp.BaseParallelApplication class method), 587
class_get_help() (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 594
class_get_help() (IPython.parallel.apps.ipclusterapp.IPClusterEngine class method), 598
class_get_help() (IPython.parallel.apps.ipclusterapp.IPClusterStart class method), 603
class_get_help() (IPython.parallel.apps.ipclusterapp.IPClusterStop class method), 609
class_get_help() (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp class method), 615
class_get_help() (IPython.parallel.apps.ipengineapp.IPEngineApp class method), 622
class_get_help() (IPython.parallel.apps.ipengineappMPI class method), 628
class_get_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp class method), 630
class_get_help() (IPython.parallel.apps.launcher.BaseLauncher class method), 636
class_get_help() (IPython.parallel.apps.launcher.BatchSystemLauncher class method), 639

class _get_help() (IPython.parallel.apps.launcher.IPClusterLauncher)
 class method), 642

class _get_help() (IPython.parallel.apps.launcher.LocalCluster)
 class method), 656

class _get_help() (IPython.parallel.apps.launcher.LocalEngines)
 class method), 659

class _get_help() (IPython.parallel.apps.launcher.LocalEnginesSet)
 class method), 662

class _get_help() (IPython.parallel.apps.launcher.LocalProcessSet)
 class method), 664

class _get_help() (IPython.parallel.apps.launcher.LSFController)
 class method), 646

class _get_help() (IPython.parallel.apps.launcher.LSFEngineSet)
 class method), 649

class _get_help() (IPython.parallel.apps.launcher.LSFLauncher)
 class method), 653

class _get_help() (IPython.parallel.apps.launcher.MPIExecutive)
 class method), 667

class _get_help() (IPython.parallel.apps.launcher.MPIExecutiveEngine)
 class method), 670

class _get_help() (IPython.parallel.apps.launcher.MPIExecutiveIsagather)
 class method), 673

class _get_help() (IPython.parallel.apps.launcher.PBSCohort)
 class method), 676

class _get_help() (IPython.parallel.apps.launcher.PBSEngineSet)
 class method), 680

class _get_help() (IPython.parallel.apps.launcher.PBSLauncher)
 class method), 683

class _get_help() (IPython.parallel.apps.launcher.SGECohort)
 class method), 687

class _get_help() (IPython.parallel.apps.launcher.SGEEngineSet)
 class method), 690

class _get_help() (IPython.parallel.apps.launcher.SGELauncher)
 class method), 694

class _get_help() (IPython.parallel.apps.launcher.SSHCohort)
 class method), 697

class _get_help() (IPython.parallel.apps.launcher.SSHEngineLauncher)
 class method), 700

class _get_help() (IPython.parallel.apps.launcher.SSHEngineSet)
 class method), 703

class _get_help() (IPython.parallel.apps.launcher.SSHLauncher)
 class method), 706

class _get_help() (IPython.parallel.apps.launcher.WindowsHPCController)
 class method), 709

class _get_help() (IPython.parallel.apps.launcher.WindowsHPCIPythonSetConfigurable)
 class method), 712

class _get_help() (IPython.parallel.apps.launcher.WindowsHPCAliasManager)
 class method), 715

class _gather_help() (IPython.parallel.apps.logwatcher.LogWatcher)
 class method), 718

class _gather_help() (IPython.parallel.apps.winhpjob.IPCControllerJob)
 class method), 722

class _gather_help() (IPython.parallel.apps.winhpjob.IPCControllerTask)
 class method), 725

class _gather_help() (IPython.parallel.apps.winhpjob.IPEngineSetJob)
 class method), 728

class _gather_help() (IPython.parallel.apps.winhpjob.IPEngineTask)
 class method), 731

class _gather_help() (IPython.parallel.apps.winhpjob.WinHPCJob)
 class method), 734

class _gather_help() (IPython.parallel.apps.winhpjob.WinHPCTask)
 class method), 737

class _gather_help() (IPython.parallel.controller.dictdb.BaseDB)
 class method), 776

class _gather_help() (IPython.parallel.controller.DictDB)
 class method), 779

class _gather_help() (IPython.parallel.controller.HeartMonitor)
 class method), 782

class _gather_help() (IPython.parallel.controller.hub.Hub)
 class method), 787

class _gather_help() (IPython.parallel.controller.hub.HubFactory)
 class method), 791

class _gather_help() (IPython.parallel.controller.TaskScheduler)
 class method), 795

class _gather_help() (IPython.parallel.controller.sqlitedb.SQLiteDB)
 class method), 800

class _gather_help() (IPython.parallel.engine.EngineFactory)
 class method), 803

class _gather_help() (IPython.parallel.engine.StreamKernel)
 class method), 808

class _gather_help() (IPython.parallel.factory.RegistrationFactory)
 class method), 823

class _get_trait_help() (IPython.config.Application)
 class method), 264

class _get_trait_help() (IPython.core.alias.AliasManager)
 class method), 269

class _get_trait_help() (IPython.config.LoggingConfigurable)
 class method), 271

class _get_trait_help() (IPython.core.singleton.SingletonConfigurable)
 class method), 273

method), 284
`class_get_trait_help()`
 (IPython.core.application.BaseIPythonApplication
 class method), 288
`class_get_trait_help()`
 (IPython.core.builtin_trap.BuiltinTrap
 class method), 294
`class_get_trait_help()`
 (IPython.core.display_trap.DisplayTrap
 class method), 317
`class_get_trait_help()`
 (IPython.core.displayhook.DisplayHook
 class method), 319
`class_get_trait_help()`
 (IPython.core.displaypub.DisplayPublisher
 class method), 323
`class_get_trait_help()`
 (IPython.core.extensions.ExtensionManager
 class method), 328
`class_get_trait_help()`
 (IPython.core.formatters.BaseFormatter
 class method), 332
`class_get_trait_help()`
 (IPython.core.formatters.DisplayFormatter
 class method), 334
`class_get_trait_help()`
 (IPython.core.formatters.HTMLFormatter
 class method), 337
`class_get_trait_help()`
 (IPython.core.formatters.JavascriptFormatter
 class method), 343
`class_get_trait_help()`
 (IPython.core.formatters.JSONFormatter
 class method), 340
`class_get_trait_help()`
 (IPython.core.formatters.LatexFormatter
 class method), 346
`class_get_trait_help()`
 (IPython.core.formatters.PlainTextFormatter
 class method), 352
`class_get_trait_help()`
 (IPython.core.formatters.PNGFormatter
 class method), 349
`class_get_trait_help()`
 (IPython.core.formatters.SVGFormatter
 class method), 355
`class_get_trait_help()`
 (IPython.core.history.HistoryManager
 class method), 359
`class_get_trait_help()`
 (IPython.core.interactiveshell.InteractiveShell
 class method), 377
`class_get_trait_help()`
 (IPython.core.payload.PayloadManager
 class method), 454
`class_get_trait_help()`
 (IPython.core.plugin.Plugin
 class method), 457
`class_get_trait_help()`
 (IPython.core.plugin.PluginManager
 class method), 459
`class_get_trait_help()`
 (IPython.core.prefilter.AliasChecker
 class method), 462
`class_get_trait_help()`
 (IPython.core.prefilter.AliasHandler
 class method), 464
`class_get_trait_help()`
 (IPython.core.prefilter.AssignMagicTransformer
 class method), 465
`class_get_trait_help()`
 (IPython.core.prefilter.AssignmentChecker
 class method), 469
`class_get_trait_help()`
 (IPython.core.prefilter.AssignSystemTransformer
 class method), 467
`class_get_trait_help()`
 (IPython.core.prefilter.AutocallChecker
 class method), 475
`class_get_trait_help()`
 (IPython.core.prefilter.AutoHandler
 class method), 471
`class_get_trait_help()`
 (IPython.core.prefilter.AutoMagicChecker
 class method), 473
`class_get_trait_help()`
 (IPython.core.prefilter.EmacsChecker
 class method), 476
`class_get_trait_help()`
 (IPython.core.prefilter.EmacsHandler
 class method), 478
`class_get_trait_help()`
 (IPython.core.prefilter.EscCharsChecker
 class method), 480
`class_get_trait_help()`
 (IPython.core.prefilter.HelpHandler
 class method), 482

class_get_trait_help()
 (IPython.core.prefilter.IPyAutocallChecker
 class method), 484

class_get_trait_help()
 (IPython.core.prefilter.IPyPromptTransformer
 class method), 486

class_get_trait_help()
 (IPython.core.prefilter.MacroChecker
 class method), 488

class_get_trait_help()
 (IPython.core.prefilter.MacroHandler
 class method), 490

class_get_trait_help()
 (IPython.core.prefilter.MagicHandler
 class method), 492

class_get_trait_help()
 (IPython.core.prefilter.MultiLineMagicChecker
 class method), 494

class_get_trait_help()
 (IPython.core.prefilter.PrefilterChecker
 class method), 495

class_get_trait_help()
 (IPython.core.prefilter.PrefilterHandler
 class method), 497

class_get_trait_help()
 (IPython.core.prefilter.PrefilterManager
 class method), 500

class_get_trait_help()
 (IPython.core.prefilter.PrefilterTransformer
 class method), 503

class_get_trait_help()
 (IPython.core.prefilter.PyPromptTransformer
 class method), 505

class_get_trait_help()
 (IPython.core.prefilter.PythonOpsChecker
 class method), 507

class_get_trait_help()
 (IPython.core.prefilter.ShellEscapeChecker
 class method), 509

class_get_trait_help()
 (IPython.core.prefilter.ShellEscapeHandler
 class method), 510

class_get_trait_help()
 (IPython.core.profileapp.ProfileApp
 class method), 513

class_get_trait_help()
 (IPython.core.profileapp.ProfileCreate
 class method), 517

class_get_trait_help()
 (IPython.core.profileapp.ProfileList
 class method), 521

class_get_trait_help()
 (IPython.core.profiledir.ProfileDir
 class method), 526

class_get_trait_help()
 (IPython.core.shellapp.InteractiveShellApp
 class method), 533

class_get_trait_help()
 (IPython.parallel.apps.baseapp.BaseParallelApplication
 class method), 587

class_get_trait_help()
 (IPython.parallel.apps.ipclusterapp.IPClusterApp
 class method), 594

class_get_trait_help()
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 class method), 598

class_get_trait_help()
 (IPython.parallel.apps.ipclusterapp.IPClusterStart
 class method), 604

class_get_trait_help()
 (IPython.parallel.apps.ipclusterapp.IPClusterStop
 class method), 609

class_get_trait_help()
 (IPython.parallel.apps.ipcontrollerapp.IPControllerApp
 class method), 615

class_get_trait_help()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 class method), 622

class_get_trait_help()
 (IPython.parallel.apps.ipengineappMPI
 class method), 628

class_get_trait_help()
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 class method), 630

class_get_trait_help()
 (IPython.parallel.apps.launcher.BaseLauncher
 class method), 636

class_get_trait_help()
 (IPython.parallel.apps.launcher.BatchSystemLauncher
 class method), 639

class_get_trait_help()
 (IPython.parallel.apps.launcher.IPClusterLauncher
 class method), 643

class_get_trait_help()
 (IPython.parallel.apps.launcher.LocalControllerLauncher
 class method), 656

```

class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LocalEngineLauncher (IPython.parallel.apps.launcher.SSHEngineLauncher
        class method), 659                           class method), 700
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LocalEngineSetLaunch (IPython.parallel.apps.launcher.SSHEngineSetLauncher
        class method), 662                           class method), 703
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LocalProcessLauncher (IPython.parallel.apps.launcher.SSHLauncher
        class method), 664                           class method), 706
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LSFControllerLaunched (IPython.parallel.apps.launcher.WindowsHPCControllerLau
        class method), 646                           class method), 709
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LSFEngineSetLaunched (IPython.parallel.apps.launcher.WindowsHPCEngineSetLau
        class method), 649                           class method), 712
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.LSFLauncher          (IPython.parallel.apps.launcher.WindowsHPCLauncher
        class method), 653                           class method), 715
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.MPIExecControllerLau (IPython.parallel.apps.logwatcher.LogWatcher
        class method), 667                           class method), 718
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.MPIExecEngineSetLaun (IPython.parallel.apps.winhpcjob.IPControllerJob
        class method), 670                           class method), 722
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.MPIExecLauncher       (IPython.parallel.apps.winhpcjob.IPControllerTask
        class method), 673                           class method), 725
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.PBSControllerLaunched (IPython.parallel.apps.winhpcjob.IPEngineSetJob
        class method), 676                           class method), 728
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.PBSEngineSetLaunched (IPython.parallel.apps.winhpcjob.IPEngineTask
        class method), 680                           class method), 731
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.PBSLauncher          (IPython.parallel.apps.winhpcjob.WinHPCJob
        class method), 683                           class method), 734
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.SGEControllerLaunced (IPython.parallel.apps.winhpcjob.WinHPCTask
        class method), 687                           class method), 737
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.SGEEngineSetLaunched (IPython.parallel.controller.dictdb.BaseDB
        class method), 690                           class method), 776
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.SGELauncher          (IPython.parallel.controller.dictdb.DictDB
        class method), 694                           class method), 779
class_get_trait_help()                                class_get_trait_help()
    (IPython.parallel.apps.launcher.SSHControllerLaunched (IPython.parallel.controller.heartmonitor.HeartMonitor
        class method), 697                           class method), 782

```

class_get_trait_help()
 (IPython.parallel.controller.hub.Hub
 class method), 787

class_get_trait_help()
 (IPython.parallel.controller.hub.HubFactory
 class method), 791

class_get_trait_help()
 (IPython.parallel.controller.scheduler.TaskScheduler
 class method), 795

class_get_trait_help()
 (IPython.parallel.controller.sqlitedb.SQLiteDB
 class method), 800

class_get_trait_help()
 (IPython.parallel.engine.engine.EngineFactory
 class method), 803

class_get_trait_help()
 (IPython.parallel.engine.streamkernel.Kernel
 class method), 808

class_get_trait_help()
 (IPython.parallel.factory.RegistrationFactory
 class method), 823

class_of() (in module IPython.utils traitlets), 934

class_print_help() (IPython.config.application.Application)
 class method), 264

class_print_help() (IPython.config.configurable.Configurable)
 class method), 269

class_print_help() (IPython.config.configurable.LoggingConfigurable)
 class method), 271

class_print_help() (IPython.config.configurable.SingletonConfigurable)
 class method), 273

class_print_help() (IPython.core.alias.AliasManager)
 class method), 284

class_print_help() (IPython.core.application.BaseIPythonApplication)
 class method), 288

class_print_help() (IPython.core.builtin_trap.BuiltinTrap)
 class method), 294

class_print_help() (IPython.core.display_trap.DisplayTrap)
 class method), 317

class_print_help() (IPython.core.displayhook.DisplayHook)
 class method), 319

class_print_help() (IPython.core.displaypub.DisplayPublisher)
 class method), 323

class_print_help() (IPython.core.extensions.ExtensionManager)
 class method), 328

class_print_help() (IPython.core.formatters.BaseFormatter)
 class method), 332

class_print_help() (IPython.core.formatters.DisplayFormatter)
 class method), 334

class_print_help() (IPython.core.formatters.HTMLFormatter
 class method), 337

class_print_help() (IPython.core.formatters.JavascriptFormatter
 class method), 343

class_print_help() (IPython.core.formatters.JSONFormatter
 class method), 340

class_print_help() (IPython.core.formatters.LatexFormatter
 class method), 346

class_print_help() (IPython.core.formatters.PlainTextFormatter
 class method), 352

class_print_help() (IPython.core.formatters.PNGFormatter
 class method), 349

class_print_help() (IPython.core.formatters.SVGFormatter
 class method), 355

class_print_help() (IPython.core.history.HistoryManager
 class method), 359

class_print_help() (IPython.core.interactiveshell.InteractiveShell
 class method), 377

class_print_help() (IPython.core.payload.PayloadManager
 class method), 454

class_print_help() (IPython.core.plugin.Plugin class
 method), 458

class_print_help() (IPython.core.plugin.PluginManager
 class method), 459

class_print_help() (IPython.core.prefilter.AliasChecker
 class method), 462

class_print_help() (IPython.core.prefilter.AliasHandler
 class method), 464

class_print_help() (IPython.core.prefilter.AssignMagicTransformer
 class method), 465

class_print_help() (IPython.core.prefilter.AssignmentChecker
 class method), 469

class_print_help() (IPython.core.prefilter.AssignSystemTransformer
 class method), 467

class_print_help() (IPython.core.prefilter.AutoHandler
 class method), 475

class_print_help() (IPython.core.prefilter.AutoHandler
 class method), 471

class_print_help() (IPython.core.prefilter.AutoMagicChecker
 class method), 473

class_print_help() (IPython.core.prefilter.EmacsChecker
 class method), 477

class_print_help() (IPython.core.prefilter.EmacsHandler
 class method), 478

class_print_help() (IPython.core.prefilter.EscCharsChecker
 class method), 480

class_print_help() (IPython.core.prefilter.HelpHandler
 class method), 482

class_print_help() (IPython.core.prefilter.IPyAutocallCheckers.print_help() (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp
 class method), 484
class_print_help() (IPython.core.prefilter.IPyPromptTransformer.print_help() (IPython.parallel.apps.ipengineapp.IPEngineApp
 class method), 486
class_print_help() (IPython.core.prefilter.MacroChecker.print_help() (IPython.parallel.apps.ipengineapp.MPI
 class method), 488
class_print_help() (IPython.core.prefilter.MacroHandle.print_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 class method), 490
class_print_help() (IPython.core.prefilter.MagicHandle.print_help() (IPython.parallel.apps.launcher.BaseLauncher
 class method), 492
class_print_help() (IPython.core.prefilter.MultiLineMagicChecker.print_help() (IPython.parallel.apps.launcher.BatchSystemLauncher
 class method), 494
class_print_help() (IPython.core.prefilter.PrefilterCheckers.print_help() (IPython.parallel.apps.launcher.IPClusterLauncher
 class method), 495
class_print_help() (IPython.core.prefilter.PrefilterHandlers.print_help() (IPython.parallel.apps.launcher.LocalControllerLauncher
 class method), 497
class_print_help() (IPython.core.prefilter.PrefilterManager.print_help() (IPython.parallel.apps.launcher.LocalEngineLauncher
 class method), 500
class_print_help() (IPython.core.prefilter.PrefilterTransformer.print_help() (IPython.parallel.apps.launcher.LocalEngineSetLauncher
 class method), 503
class_print_help() (IPython.core.prefilter.PyPromptTransformer.print_help() (IPython.parallel.apps.launcher.LocalProcessLauncher
 class method), 505
class_print_help() (IPython.core.prefilter.PythonOpsCheckers.print_help() (IPython.parallel.apps.launcher.LSFControllerLauncher
 class method), 507
class_print_help() (IPython.core.prefilter.ShellEscapeCheckers.print_help() (IPython.parallel.apps.launcher.LSFEngineSetLauncher
 class method), 509
class_print_help() (IPython.core.prefilter.ShellEscapeHandlers.print_help() (IPython.parallel.apps.launcher.LSFLauncher
 class method), 510
class_print_help() (IPython.core.profileapp.ProfileApp.print_help() (IPython.parallel.apps.launcherMPIExecController
 class method), 513
class_print_help() (IPython.core.profileapp.ProfileCreate.print_help() (IPython.parallel.apps.launcherMPIExecEngineSetLauncher
 class method), 517
class_print_help() (IPython.core.profileapp.ProfileList.print_help() (IPython.parallel.apps.launcherMPIExecLauncher
 class method), 521
class_print_help() (IPython.core.profiledir.ProfileDir.print_help() (IPython.parallel.apps.launcher.PBSControllerLauncher
 class method), 526
class_print_help() (IPython.core.shellapp.InteractiveShellApp.print_help() (IPython.parallel.apps.launcher.PBSEngineSetLauncher
 class method), 533
class_print_help() (IPython.parallel.apps.baseapp.BaseParallelApp.print_help() (IPython.parallel.apps.launcher.PBSLauncher
 class method), 587
class_print_help() (IPython.parallel.apps.ipclusterapp.IPClusterApp.print_help() (IPython.parallel.apps.launcher.SGEControllerLauncher
 class method), 594
class_print_help() (IPython.parallel.apps.ipclusterapp.IPClusterEngines.print_help() (IPython.parallel.apps.launcher.SGEEngineSetLauncher
 class method), 598
class_print_help() (IPython.parallel.apps.ipclusterapp.IPClusterStart.print_help() (IPython.parallel.apps.launcher.SGELauncher
 class method), 604
class_print_help() (IPython.parallel.apps.ipclusterapp.IPClusterStop.print_help() (IPython.parallel.apps.launcher.SSHControllerLauncher
 class method), 609
class_print_help() (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp.print_help() (IPython.parallel.apps.launcher.IPCControllerApp
 class method), 615
class_print_help() (IPython.parallel.apps.ipengineapp.IPEngineApp.print_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 class method), 622
class_print_help() (IPython.parallel.apps.ipengineapp.MPI.print_help() (IPython.parallel.apps.launcherBaseLauncher
 class method), 628
class_print_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp.print_help() (IPython.parallel.apps.launcherLocalControllerLauncher
 class method), 630
class_print_help() (IPython.parallel.apps.launcherBaseLauncher.print_help() (IPython.parallel.apps.launcherLocalControllerLauncher
 class method), 636
class_print_help() (IPython.parallel.apps.launcherBatchSystemLauncher.print_help() (IPython.parallel.apps.launcherLocalEngineLauncher
 class method), 639
class_print_help() (IPython.parallel.apps.launcherIPClusterLauncher.print_help() (IPython.parallel.apps.launcherLocalEngineSetLauncher
 class method), 643
class_print_help() (IPython.parallel.apps.launcherLocalControllerLauncher.print_help() (IPython.parallel.apps.launcherLocalProcessLauncher
 class method), 656
class_print_help() (IPython.parallel.apps.launcherLocalEngineLauncher.print_help() (IPython.parallel.apps.launcherLSFControllerLauncher
 class method), 659
class_print_help() (IPython.parallel.apps.launcherLocalEngineSetLauncher.print_help() (IPython.parallel.apps.launcherLSFEngineSetLauncher
 class method), 662
class_print_help() (IPython.parallel.apps.launcherLocalProcessLauncher.print_help() (IPython.parallel.apps.launcherLSFLauncher
 class method), 664
class_print_help() (IPython.parallel.apps.launcherLSFControllerLauncher.print_help() (IPython.parallel.apps.launcherLSFEngineSetLauncher
 class method), 666
class_print_help() (IPython.parallel.apps.launcherLSFEngineSetLauncher.print_help() (IPython.parallel.apps.launcherLSFLauncher
 class method), 669
class_print_help() (IPython.parallel.apps.launcherMPIExecController.print_help() (IPython.parallel.apps.launcherMPIExecEngineSetLauncher
 class method), 667
class_print_help() (IPython.parallel.apps.launcherMPIExecEngineSetLauncher.print_help() (IPython.parallel.apps.launcherMPIExecLauncher
 class method), 670
class_print_help() (IPython.parallel.apps.launcherMPIExecLauncher.print_help() (IPython.parallel.apps.launcherMPIExecSetLauncher
 class method), 673
class_print_help() (IPython.parallel.apps.launcherPBSCControllerLauncher.print_help() (IPython.parallel.apps.launcherPBSEngineSetLauncher
 class method), 676
class_print_help() (IPython.parallel.apps.launcherInteractiveShellApp.print_help() (IPython.parallel.apps.launcherPBSEngineSetLauncher
 class method), 680
class_print_help() (IPython.parallel.apps.launcherParallelApp.print_help() (IPython.parallel.apps.launcherPBSLauncher
 class method), 683
class_print_help() (IPython.parallel.apps.launcherSGEControllerLauncher.print_help() (IPython.parallel.apps.launcherSGEEngineSetLauncher
 class method), 687
class_print_help() (IPython.parallel.apps.launcherSGEEngineSetLauncher.print_help() (IPython.parallel.apps.launcherSGELauncher
 class method), 690
class_print_help() (IPython.parallel.apps.launcherSGELauncher.print_help() (IPython.parallel.apps.launcherSSHControllerLauncher
 class method), 694
class_print_help() (IPython.parallel.apps.launcherSSHControllerLauncher.print_help() (IPython.parallel.apps.launcherSSHLauncher
 class method), 697

class_print_help() (IPython.parallel.apps.launcher.SSHEngine trait_names) (IPython.config.configurable.Configurable
 class method), 700
class_print_help() (IPython.parallel.apps.launcher.SSHEngine trait_names) (IPython.config.configurable.LoggingConfigurable
 class method), 269
class_print_help() (IPython.parallel.apps.launcher.SSHEngine trait_names) (IPython.config.configurable.LoggingConfigurable
 class method), 271
class_print_help() (IPython.parallel.apps.launcher.SSHEngine trait_names) (IPython.config.configurable.SingletonConfigurable
 class method), 273
class_print_help() (IPython.parallel.apps.launcher.WindowsHPGController trait_names) (IPython.core.alias.AliasManager
 class method), 284
class_print_help() (IPython.parallel.apps.launcher.WindowsHPGController trait_names) (IPython.core.application.BaseIPythonApplication
 class method), 288
class_print_help() (IPython.parallel.apps.launcher.WindowsHPGLauncher trait_names) (IPython.core.builtin_trap.BuiltinTrap
 class method), 294
class_print_help() (IPython.parallel.apps.logwatcher.LogWatcher trait_names) (IPython.core.display_trap.DisplayTrap
 class method), 317
class_print_help() (IPython.parallel.apps.winhpcjob.IPClusterJob trait_names) (IPython.core.displayhook.DisplayHook
 class method), 319
class_print_help() (IPython.parallel.apps.winhpcjob.IPClusterTask trait_names) (IPython.core.displaypub.DisplayPublisher
 class method), 323
class_print_help() (IPython.parallel.apps.winhpcjob.IPEngineJob trait_names) (IPython.core.extensions.ExtensionManager
 class method), 328
class_print_help() (IPython.parallel.apps.winhpcjob.IPEngineTask trait_names) (IPython.core.formatters.BaseFormatter
 class method), 332
class_print_help() (IPython.parallel.apps.winhpcjob.WiHPCFault trait_names) (IPython.core.formatters.DisplayFormatter
 class method), 334
class_print_help() (IPython.parallel.apps.winhpcjob.WiHPCTask trait_names) (IPython.core.formatters.HTMLFormatter
 class method), 337
class_print_help() (IPython.parallel.controller.dictdb.BakedDB trait_names) (IPython.core.formatters.JavascriptFormatter
 class method), 343
class_print_help() (IPython.parallel.controller.dictdb.DictDB trait_names) (IPython.core.formatters.JSONFormatter
 class method), 340
class_print_help() (IPython.parallel.controller.heartmonitor.HeartMonitor trait_names) (IPython.core.formatters.LatexFormatter
 class method), 346
class_print_help() (IPython.parallel.controller.hub.Hub trait_names) (IPython.core.formatters.PlainTextFormatter
 class method), 352
class_print_help() (IPython.parallel.controller.hub.Hub trait_names) (IPython.core.formatters.PNGFormatter
 class method), 349
class_print_help() (IPython.parallel.controller.scheduler.TaskScheduler trait_names) (IPython.core.formatters.SVGFormatter
 class method), 355
class_print_help() (IPython.parallel.controller.sqlitedb.SQLiteDB trait_names) (IPython.core.history.HistoryManager
 class method), 359
class_print_help() (IPython.parallel.engine.Engine trait_names) (IPython.core.interactiveshell.InteractiveShell
 class method), 377
class_print_help() (IPython.parallel.engine.streamkernel.Kernel trait_names) (IPython.core.payload.PayloadManager
 class method), 454
class_print_help() (IPython.parallel.factory.RegistrationFactory trait_names) (IPython.core.plugin.Plugin
 class method), 458
class_trait_names() (IPython.config.application.Application trait_names) (IPython.core.plugin.PluginManager
 class method), 459

class_trait_names() (IPython.core.prefilter.AliasCheckersTraitNames) (IPython.core.prefilter.ShellEscapeChecker class method), 462
class_trait_names() (IPython.core.prefilter.AliasHandlerTraitNames) (IPython.core.prefilter.ShellEscapeHandler class method), 464
class_trait_names() (IPython.core.prefilter.AssignMagicTraitNames) (IPython.core.profileapp.ProfileApp class method), 466
class_trait_names() (IPython.core.prefilter.AssignmentClassTraitNames) (IPython.core.profileapp.ProfileCreate class method), 469
class_trait_names() (IPython.core.prefilter.AssignSystemTraitNames) (IPython.core.profileapp.ProfileList class method), 467
class_trait_names() (IPython.core.prefilter.AutocallCheckersTraitNames) (IPython.core.profiledir.ProfileDir class method), 475
class_trait_names() (IPython.core.prefilter.AutoHandleTraitNames) (IPython.core.shellapp.InteractiveShellApp class method), 471
class_trait_names() (IPython.core.prefilter.AutoMagicCheckersTraitNames) (IPython.parallel.apps.baseapp.BaseParallelApp class method), 473
class_trait_names() (IPython.core.prefilter.EmacsCheckersTraitNames) (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 477
class_trait_names() (IPython.core.prefilter.EmacsHandlerTraitNames) (IPython.parallel.apps.ipclusterapp.IPClusterEngine class method), 478
class_trait_names() (IPython.core.prefilter.EscCharsCheckersTraitNames) (IPython.parallel.apps.ipclusterapp.IPClusterStar class method), 480
class_trait_names() (IPython.core.prefilter.HelpHandleTraitNames) (IPython.parallel.apps.ipclusterapp.IPClusterStop class method), 482
class_trait_names() (IPython.core.prefilter.IPyAutocallCheckersTraitNames) (IPython.parallel.apps.ipcontrollerapp.IPController class method), 484
class_trait_names() (IPython.core.prefilter.IPyPromptTransformerTraitNames) (IPython.parallel.apps.ipengineapp.IPEngineApp class method), 486
class_trait_names() (IPython.core.prefilter.MacroCheckersTraitNames) (IPython.parallel.apps.ipengineapp.MPI class method), 488
class_trait_names() (IPython.core.prefilter.MacroHandlerTraitNames) (IPython.parallel.apps.iploggerapp.IPLoaderApp class method), 490
class_trait_names() (IPython.core.prefilter.MagicHandlerTraitNames) (IPython.parallel.apps.launcher.BaseLauncher class method), 492
class_trait_names() (IPython.core.prefilter.MultiLineMagicCheckersTraitNames) (IPython.parallel.apps.launcher.BatchSystemLauncher class method), 494
class_trait_names() (IPython.core.prefilter.PrefilterCheckersTraitNames) (IPython.parallel.apps.launcher.IPClusterLauncher class method), 496
class_trait_names() (IPython.core.prefilter.PrefilterHandlerTraitNames) (IPython.parallel.apps.launcher.LocalControllerLauncher class method), 498
class_trait_names() (IPython.core.prefilter.PrefilterManagerTraitNames) (IPython.parallel.apps.launcher.LocalEngineLauncher class method), 500
class_trait_names() (IPython.core.prefilter.PrefilterTransformerTraitNames) (IPython.parallel.apps.launcher.LocalEngineSetLauncher class method), 503
class_trait_names() (IPython.core.prefilter.PyPromptTransformerTraitNames) (IPython.parallel.apps.launcher.LocalProcessLauncher class method), 505
class_trait_names() (IPython.core.prefilter.PythonOpsCheckersTraitNames) (IPython.parallel.apps.launcher.LSFControllerLauncher class method), 507

class_trait_names() (IPython.parallel.apps.launcher.LSFEngineSetNames) (IPython.parallel.apps.winhpcjob.WinhPCTask
 class method), 649
 class method), 737

class_trait_names() (IPython.parallel.apps.launcher.LSFEngineSetNames) (IPython.parallel.client.client.Client
 class method), 653
 class method), 746

class_trait_names() (IPython.parallel.apps.launcher.MRIExecController) (IPython.parallel.client.view.DirectView
 class method), 667
 class method), 757

class_trait_names() (IPython.parallel.apps.launcher.MRIExecEngineSetNames) (IPython.parallel.client.view.LoadBalancedView
 class method), 670
 class method), 763

class_trait_names() (IPython.parallel.apps.launcher.MRIExecEnvironments) (IPython.parallel.client.view.View
 class method), 673
 class method), 768

class_trait_names() (IPython.parallel.apps.launcher.PBSControllerNames) (Python.parallel.controller.dictdb.BaseDB
 class method), 676
 class method), 776

class_trait_names() (IPython.parallel.apps.launcher.PBSEngineSetNames) (Python.parallel.controller.dictdb.DictDB
 class method), 680
 class method), 779

class_trait_names() (IPython.parallel.apps.launcher.PBSHasTraitNames) (IPython.parallel.controller.heartmonitor.HeartM
 class method), 683
 class method), 782

class_trait_names() (IPython.parallel.apps.launcher.SGEControllerNames) (Python.parallel.controller.hub.EngineConnector
 class method), 687
 class method), 784

class_trait_names() (IPython.parallel.apps.launcher.SGEEngineSetNames) (Python.parallel.controller.hub.Hub
 class method), 690
 class method), 787

class_trait_names() (IPython.parallel.apps.launcher.SGEHasTraitNames) (IPython.parallel.controller.hub.HubFactory
 class method), 694
 class method), 791

class_trait_names() (IPython.parallel.apps.launcher.SSHControllerNames) (Python.parallel.controller.scheduler.TaskSchedu
 class method), 697
 class method), 796

class_trait_names() (IPython.parallel.apps.launcher.SSHEngineSetNames) (IPython.parallel.controller.sqlitedb.SQLiteDB
 class method), 700
 class method), 801

class_trait_names() (IPython.parallel.apps.launcher.SSHHasTraitNames) (Python.parallel.engine.engine.EngineFactory
 class method), 703
 class method), 803

class_trait_names() (IPython.parallel.apps.launcher.SSHLaunchNames) (IPython.parallel.engine.streamkernel.Kernel
 class method), 706
 class method), 809

class_trait_names() (IPython.parallel.apps.launcher.WindowSHPCComms) (IPython.parallel.factory.RegistrationFactory
 class method), 709
 class method), 824

class_trait_names() (IPython.parallel.apps.launcher.WindowSHPCEngines) (IPython.parallel.utils.traitlets.HasTraits
 class method), 712
 class method), 919

class_trait_names() (IPython.parallel.apps.launcher.WindowSHPCQL) (IPython.config.application.Application
 class method), 715
 class method), 264

class_trait_names() (IPython.parallel.apps.logwatcher.IctusWatcher) (IPython.config.configurable.Configurable
 class method), 718
 class method), 269

class_trait_names() (IPython.parallel.apps.winhpcjob.IPCController) (IPython.config.configurable.LoggingConfigurable
 class method), 722
 class method), 271

class_trait_names() (IPython.parallel.apps.winhpcjob.IPCController) (IPython.config.configurable.SingletonConfigurable
 class method), 725
 class method), 273

class_trait_names() (IPython.parallel.apps.winhpcjob.IPEngineSet) (IPython.core.alias.AliasManager class
 class method), 728
 method), 284

class_trait_names() (IPython.parallel.apps.winhpcjob.IPEngineTask) (IPython.core.application.BaseIPythonApplication
 class method), 731
 class method), 288

class_trait_names() (IPython.parallel.apps.winhpcjob.WhishPCJob) (IPython.core.builtin_trap.BuiltinTrap
 class method), 734
 class method), 295

class_traits() (IPython.core.shellapp.InteractiveShellApp) (IPython.parallel.apps.launcher.PBSEngineSetLauncher
 class method), 533
 class method), 680
class_traits() (IPython.parallel.apps.baseapp.BaseParallelApp) (IPython.parallel.apps.launcher.PBSLauncher
 class method), 588
 class method), 683
class_traits() (IPython.parallel.apps.ipclusterapp.IPClusterApp) (IPython.parallel.apps.launcher.SGEControllerLauncher
 class method), 594
 class method), 687
class_traits() (IPython.parallel.apps.ipclusterapp.IPClusterEngines) (IPython.parallel.apps.launcher.SGEEngineSetLauncher
 class method), 598
 class method), 690
class_traits() (IPython.parallel.apps.ipclusterapp.IPClusterStart) (IPython.parallel.apps.launcher.SGELauncher
 class method), 604
 class method), 694
class_traits() (IPython.parallel.apps.ipclusterapp.IPClusterStop) (IPython.parallel.apps.launcher.SSHControllerLauncher
 class method), 610
 class method), 697
class_traits() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp) (IPython.parallel.apps.launcher.SSHEngineLauncher
 class method), 616
 class method), 700
class_traits() (IPython.parallel.apps.ipengineapp.IPEngineApp) (IPython.parallel.apps.launcher.SSHEngineSetLauncher
 class method), 622
 class method), 703
class_traits() (IPython.parallel.apps.ipengineapp.MPI) (IPython.parallel.apps.launcher.SSHLauncher
 class method), 628
 class method), 706
class_traits() (IPython.parallel.apps.iploggerapp.IPLoggerApp) (IPython.parallel.apps.launcher.WindowsHPCController
 class method), 630
 class method), 709
class_traits() (IPython.parallel.apps.launcher.BaseLauncher) (IPython.parallel.apps.launcher.WindowsHPCEngineSet
 class method), 637
 class method), 712
class_traits() (IPython.parallel.apps.launcher.BatchSystem) (IPython.parallel.apps.launcher.WindowsHPCLauncher
 class method), 640
 class method), 715
class_traits() (IPython.parallel.apps.launcher.IPClusterController) (IPython.parallel.apps.logwatcher.LogWatcher
 class method), 643
 class method), 718
class_traits() (IPython.parallel.apps.launcher.LocalController) (IPython.parallel.apps.winhpcjob.IPControllerJob
 class method), 656
 class method), 722
class_traits() (IPython.parallel.apps.launcher.LocalEngines) (IPython.parallel.apps.winhpcjob.IPControllerTask
 class method), 659
 class method), 725
class_traits() (IPython.parallel.apps.launcher.LocalEnginesSet) (IPython.parallel.apps.winhpcjob.IPEngineSetJob
 class method), 662
 class method), 728
class_traits() (IPython.parallel.apps.launcher.LocalProcess) (IPython.parallel.apps.winhpcjob.IPEngineTask
 class method), 665
 class method), 731
class_traits() (IPython.parallel.apps.launcher.LSFController) (IPython.parallel.apps.winhpcjob.WinHPCJob
 class method), 666
 class method), 734
class_traits() (IPython.parallel.apps.launcher.LSFEngines) (IPython.parallel.apps.winhpcjob.WinHPCTask
 class method), 649
 class method), 737
class_traits() (IPython.parallel.apps.launcher.LSFLaunchers) (IPython.parallel.client.client.Client
 class method), 653
 class method), 747
class_traits() (IPython.parallel.apps.launcherMPIExecClass) (IPython.parallel.client.view.DirectView
 class method), 667
 class method), 757
class_traits() (IPython.parallel.apps.launcherMPIExecEngines) (IPython.parallel.client.view.LoadBalancedView
 class method), 670
 class method), 763
class_traits() (IPython.parallel.apps.launcherMPIExecEngines) (IPython.parallel.client.view.View
 class method), 673
 class method), 768
class_traits() (IPython.parallel.apps.launcher.PBSController) (IPython.parallel.controller.dictdb.BaseDB
 class method), 676
 class method), 777

class_traits() (IPython.parallel.controller.dictdb.DictDB class method), 779
class_traits() (IPython.parallel.controller.heartmonitor.HeartMonitor class method), 782
class_traits() (IPython.parallel.controller.hub.EngineCoordinator class method), 785
class_traits() (IPython.parallel.controller.hub.Hub class method), 787
class_traits() (IPython.parallel.controller.hub.HubFactory class method), 791
class_traits() (IPython.parallel.controller.scheduler.TaskScheduler class method), 796
class_traits() (IPython.parallel.controller.sqlitedb.SQLiteDB class method), 801
class_traits() (IPython.parallel.engine.engine.EngineFactory class method), 803
class_traits() (IPython.parallel.engine.streamkernel.KernelManager class method), 809
class_traits() (IPython.parallel.factory.RegistrationFactory class method), 824
class_traits() (IPython.utils.traitlets.HasTraits class method), 919
ClassBasedTraitType (class in IPython.utils.traitlets), 912
classes (IPython.config.application.Application attribute), 264
classes (IPython.core.application.BaseIPythonApplication attribute), 288
classes (IPython.core.profileapp.ProfileApp attribute), 513
classes (IPython.core.profileapp.ProfileCreate attribute), 517
classes (IPython.core.profileapp.ProfileList attribute), 522
classes (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 588
classes (IPython.parallel.apps.ipclusterapp.IPClusterApplier attribute), 594
classes (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 598
classes (IPython.parallel.apps.ipclusterapp.IPClusterStartApplier attribute), 604
classes (IPython.parallel.apps.ipclusterapp.IPClusterStopApplier attribute), 610
class_traits() (IPython.parallel.apps.ipcontrollerapp.IPClusterController class method), 616
clean_logs (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 604
clean_logs (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 610
clear (IPython.parallel.client.client.Metadata attribute), 751
clear (IPython.parallel.controller.dependency.Dependency attribute), 773
clear (IPython.parallel.util.Namespace attribute), 826
clear (IPython.parallel.util.ReverseDict attribute), 827
clear (IPython.utils.coloransi.ColorSchemeTable attribute), 855
clear (IPython.utils.ipstruct.Struct attribute), 868
clear() (IPython.config.loader.ArgParseConfigLoader method), 277
clear() (IPython.config.loader.CommandLineConfigLoader method), 278
clear() (IPython.config.loader.ConfigLoader method), 280
clear() (IPython.config.loader.FileConfigLoader method), 281
clear() (IPython.config.loader.KeyValueConfigLoader method), 282
clear() (IPython.config.loader.PyFileConfigLoader method), 283
clear() (IPython.parallel.client.client.Client method), 747
clear() (IPython.parallel.client.view.DirectView method), 758
clear() (IPython.testing.globalipapp.ipnsdict method), 833

clear() (IPython.utils.pickleshare.PickleShareDB method), 881
clear_aliases() (IPython.core.alias.AliasManager method), 284
clear_all_breaks() (IPython.core.debugger.Pdb method), 307
clear_all_file_breaks() (IPython.core.debugger.Pdb method), 307
clear_app_refs() (IPython.lib.inpthook.InputHookManager method), 570
clear_bpbynumber() (IPython.core.debugger.Pdb method), 307
clear_break() (IPython.core.debugger.Pdb method), 307
clear_err_state() (IPython.core.ultratb.SyntaxTB method), 543
clear_inpthook() (IPython.lib.inpthook.InputHookManager method), 570
clear_instance() (IPython.config.application.Application class method), 264
clear_instance() (IPython.config.configurable.SingletonConfigurable class method), 274
clear_instance() (IPython.core.application.BaseIPythonApplication class method), 288
clear_instance() (IPython.core.interactiveshell.InteractiveShell class method), 377
clear_instance() (IPython.core.profileapp.ProfileApp class method), 513
clear_instance() (IPython.core.profileapp.ProfileCreateclients class method), 517
clear_instance() (IPython.core.profileapp.ProfileList class method), 522
clear_instance() (IPython.parallel.apps.baseapp.BaseParallelApp class method), 588
clear_instance() (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 594
clear_instance() (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 598
clear_instance() (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 604
clear_instance() (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 610
clear_instance() (IPython.parallel.apps.ipcontrollerapp.IRCControllerApp class method), 616
clear_instance() (IPython.parallel.apps.ipengineapp.IPEngineApp class method), 622
clear_instance() (IPython.parallel.apps.iploggerapp.IPLoggerApp class method), 631
clear_main_mod_cache() (IPython.core.interactiveshell.InteractiveShell method), 377
clear_payload() (IPython.core.payload.PayloadManager method), 455
clear_request() (IPython.parallel.engine.streamkernel.Kernel method), 809
ClearDemo (class in IPython.lib.demo), 557
ClearIPDemo (class in IPython.lib.demo), 559
ClearMixin (class in IPython.lib.demo), 560
Client (class in IPython.parallel.client.client), 744
client (IPython.parallel.client.view.DirectView attribute), 758
client (IPython.parallel.client.view.LoadBalancedView attribute), 763
client (IPython.parallel.client.view.View attribute), 768
client (IPython.parallel.engine.streamkernel.Kernel attribute), 809
client_info (IPython.parallel.controller.hub.Hub attribute), 787
client_ip (IPython.parallel.controller.hub.HubFactory attribute), 792
client_stream (IPython.parallel.controller.scheduler.TaskScheduler attribute), 796
client_transport (IPython.parallel.controller.hub.HubFactory attribute), 792
ClientError (class in IPython.parallel.error), 813
clients (IPython.parallel.controller.scheduler.TaskScheduler attribute), 796
CLong (class in IPython.utils.traits), 909
CLong (IPython.lib.irunner.InteractiveRunner method), 575
clients (IPython.parallel.controller.hub.Hub attribute), 367
CLogger (IPython.lib.irunner.IPythonRunner method), 574
clients (IPython.parallel.controller.hub.Hub attribute), 576
CLogger (IPython.lib.irunner.PythonRunner method), 578
clients (IPython.parallel.controller.hub.Hub attribute), 574
CLogger (IPython.lib.irunner.SAGERunner method), 578
clients (IPython.parallel.controller.hub.Hub attribute), 747
CLogger (IPython.testing.globalipapp.StreamProxy method), 833
clients (IPython.testing.mkdoctests.IndentOut method), 839

close() (IPython.utils.io.IOStream method), 865
close() (IPython.utils.io.Tee method), 866
close_log() (IPython.core.logger.Logger method), 418
closed (IPython.testing.globalipapp.StreamProxy attribute), 833
closed (IPython.utils.io.IOStream attribute), 865
cmd_and_args (IPython.parallel.apps.launcher.IPClusterLauncher method), 543
attribute), 643
cmd_and_args (IPython.parallel.apps.launcher.LocalControllerLauncher method), 545
attribute), 657
cmd_and_args (IPython.parallel.apps.launcher.LocalEngineLauncher method), 546
attribute), 659
cmd_and_args (IPython.parallel.apps.launcher.LocalProcessLauncher method), 378
attribute), 665
cmd_and_args (IPython.parallel.apps.launcher.MPIExecLauncher method), 855
attribute), 667
cmd_and_args (IPython.parallel.apps.launcher.MPIExecLaunchers class in IPython.core.ultratb), 539
attribute), 671
cmd_and_args (IPython.parallel.apps.launcher.MPIExecLaunchers class in IPython.core.ultratb), 307
attribute), 673
cmd_and_args (IPython.parallel.apps.launcher.SSHControllerLauncher method), 725
attribute), 697
cmd_and_args (IPython.parallel.apps.launcher.SSHEngineLauncher method), 731
attribute), 700
cmd_and_args (IPython.parallel.apps.launcher.SSHLauncher attribute), 737
attribute), 706
cmdloop() (IPython.core.debugger.Pdb method), 307
code (IPython.core.inputsplitter.InputSplitter attribute), 371
code (IPython.core.inputsplitter.IPythonInputSplitter attribute), 369
code_ctor() (in module IPython.utils.codeutil), 854
code_name() (in module IPython.core.compilerop), 297
code_to_run (IPython.core.shellapp.InteractiveShellApp attribute), 534
coerce_str() (IPython.utils.traits.DottedObjectName method), 916
coerce_str() (IPython.utils.traits.ObjectName method), 925
collect_exceptions() (in module IPython.parallel.error), 823
color_info (IPython.core.interactiveshell.InteractiveShell attribute), 378
color_toggle() (IPython.core.ultratb.AutoFormattedTB method), 537
color_toggle() (IPython.core.ultratb.ColorTB method), 539
color_toggle() (IPython.core.ultratb.FormattedTB method), 540
color_toggle() (IPython.core.ultratb.ListTB method), 542
color_toggle() (IPython.core.ultratb.SyntaxTB method), 543
color_toggle() (IPython.core.ultratb.TBTools method), 545
color_toggle() (IPython.core.ultratb.VerboseTB method), 546
colors (IPython.core.interactiveshell.InteractiveShell ColorScheme (class in IPython.utils.coloransi), 855
attribute), 855
ColorScheme (class in IPython.utils.coloransi), 855
command_line (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 725
command_line (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 731
command_line (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 737
CommandChainDispatcher (class in IPython.core.hooks), 367
CommandLineConfigLoader (class in IPython.config.loader), 278
commands_resuming (IPython.core.debugger.Pdb attribute), 307
compiler (IPython.parallel.engine.streamkernel.Kernel attribute), 809
compiler_flags (IPython.core.compilerop.CachingCompiler attribute), 297
complete() (IPython.core.completer.Completer method), 299
complete() (IPython.core.completer.IPCompleter method), 301
complete() (IPython.core.debugger.Pdb method), 307
complete() (IPython.core.interactiveshell.InteractiveShell method), 378
complete() (IPython.parallel.engine.streamkernel.Kernel method), 809
complete_help() (IPython.core.debugger.Pdb method), 307

complete_object() (in module IPython.utils.generics), 862

complete_registration() (IPython.parallel.engine.EngineFactory method), 804

complete_request() (IPython.parallel.engine.streamkernel.Kernel attribute), 809

completed (IPython.parallel.controller.hub.Hub attribute), 787

completed (IPython.parallel.controller.scheduler.TaskScheduler attribute), 796

completedefault() (IPython.core.debugger.Pdb method), 307

completenames() (IPython.core.debugger.Pdb method), 307

Completer (class in IPython.core.completer), 298

completer (IPython.parallel.engine.streamkernel.Kernel attribute), 809

CompletionSplitter (class in IPython.core.completer), 299

Complex (class in IPython.utils.traits), 913

CompositeError (class in IPython.parallel.error), 813

CompositeFilter (class in IPython.parallel.controller.dictdb), 778

compress_dhist() (in module IPython.core.magic), 444

compress_user() (in module IPython.core.completer), 302

compute_format_data() (IPython.core.displayhook.DisplayHook method), 319

concatenate() (IPython.parallel.client.map.Map method), 753

concatenate() (IPython.parallel.client.map.RoundRobinMap method), 753

Config (class in IPython.config.loader), 279

config (IPython.config.application.Application attribute), 264

config (IPython.config.configurable.Configurable attribute), 269

config (IPython.config.configurable.LoggingConfigurable attribute), 271

config (IPython.config.configurable.SingletonConfigurable attribute), 274

config (IPython.core.alias.AliasManager attribute), 285

config (IPython.core.application.BaseIPythonApplication attribute), 288

module config (IPython.core.builtin_trap.BuiltinTrap attribute), 295

config (IPython.core.display_trap.DisplayTrap attribute), 317

config (IPython.core.displayhook.DisplayHook attribute), 320

config (IPython.core.displaypub.DisplayPublisher attribute), 323

config (IPython.core.extensions.ExtensionManager attribute), 329

config (IPython.core.formatters.BaseFormatter attribute), 332

config (IPython.core.formatters.DisplayFormatter attribute), 335

config (IPython.core.formatters.HTMLFormatter attribute), 338

config (IPython.core.formatters.JavascriptFormatter attribute), 343

config (IPython.core.formatters.JSONFormatter attribute), 341

config (IPython.core.formatters.LatexFormatter attribute), 346

config (IPython.core.formatters.PlainTextFormatter attribute), 352

config (IPython.core.formatters.PNGFormatter attribute), 349

config (IPython.core.formatters.SVGFormatter attribute), 356

config (IPython.core.history.HistoryManager attribute), 359

config (IPython.core.interactiveshell.InteractiveShell attribute), 379

config (IPython.core.payload.PayloadManager attribute), 455

config (IPython.core.plugin.Plugin attribute), 458

config (IPython.core.plugin.PluginManager attribute), 459

config (IPython.core.prefilter.AliasChecker attribute), 462

config (IPython.core.prefilter.AliasHandler attribute), 464

config (IPython.core.prefilter.AssignMagicTransformer attribute), 466

config (IPython.core.prefilter.AssignmentChecker attribute), 470

config (IPython.core.prefilter.AssignSystemTransformer attribute), 468

config (IPython.core.prefilter.AutocallChecker attribute), 470

| | | |
|--|--|--|
| tribute), 475 | | 527 |
| config (IPython.core.prefilter.AutoHandler attribute), 471 | at- | config (IPython.core.shellapp.InteractiveShellApp attribute), 534 |
| config (IPython.core.prefilter.AutoMagicChecker attribute), 473 | at- | config (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 588 |
| config (IPython.core.prefilter.EmacsChecker attribute), 477 | at- | config (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 594 |
| config (IPython.core.prefilter.EmacsHandler attribute), 479 | at- | config (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 598 |
| config (IPython.core.prefilter.EscCharsChecker attribute), 480 | at- | config (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 604 |
| config (IPython.core.prefilter.HelpHandler attribute), 482 | at- | config (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 610 |
| config (IPython.core.prefilter.IPyAutocallChecker attribute), 484 | | config (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 616 |
| config (IPython.core.prefilter.IPyPromptTransformer attribute), 486 | | config (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623 |
| config (IPython.core.prefilter.MacroChecker attribute), 489 | at- | config (IPython.parallel.apps.ipengineappMPI attribute), 628 |
| config (IPython.core.prefilter.MacroHandler attribute), 490 | at- | config (IPython.parallel.apps.iploggerapp.IPLoggerApp attribute), 631 |
| config (IPython.core.prefilter.MagicHandler attribute), 492 | at- | config (IPython.parallel.apps.launcher.BaseLauncher attribute), 637 |
| config (IPython.core.prefilter.MultiLineMagicChecker attribute), 494 | config (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 640 | |
| config (IPython.core.prefilter.PrefilterChecker attribute), 496 | | config (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 643 |
| config (IPython.core.prefilter.PrefilterHandler attribute), 498 | | config (IPython.parallel.apps.launcher.LocalControllerLauncher attribute), 657 |
| config (IPython.core.prefilter.PrefilterManager attribute), 500 | | config (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 659 |
| config (IPython.core.prefilter.PrefilterTransformer attribute), 503 | | config (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 662 |
| config (IPython.core.prefilter.PyPromptTransformer attribute), 505 | | config (IPython.parallel.apps.launcher.LocalProcessLauncher attribute), 665 |
| config (IPython.core.prefilter.PythonOpsChecker attribute), 507 | | config (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 646 |
| config (IPython.core.prefilter.ShellEscapeChecker attribute), 509 | | config (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 650 |
| config (IPython.core.prefilter.ShellEscapeHandler attribute), 511 | | config (IPython.parallel.apps.launcher.LSFLauncher attribute), 653 |
| config (IPython.core.profileapp.ProfileApp attribute), 513 | at- | config (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 668 |
| config (IPython.core.profileapp.ProfileCreate attribute), 517 | at- | config (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 671 |
| config (IPython.core.profileapp.ProfileList attribute), 522 | at- | config (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 674 |
| config (IPython.core.profiledir.ProfileDir attribute), | | config (IPython.parallel.apps.launcher.PBSCControllerLauncher |

attribute), 677
config (IPython.parallel.apps.launcher.PBSEngineSetLauncher config (IPython.parallel.controller.TaskScheduler attribute), 680 attribute), 796
config (IPython.parallel.apps.launcher.PBSLauncher config (IPython.parallel.controller.sqlitedb.SQLiteDB attribute), 684 attribute), 801
config (IPython.parallel.apps.launcher.SGEControllerLauncher config (IPython.parallel.engine.EngineFactory attribute), 687 attribute), 804
config (IPython.parallel.apps.launcher.SGEEngineSetLauncher config (IPython.parallel.engine.streamkernel.Kernel attribute), 691 attribute), 809
config (IPython.parallel.apps.launcher.SGELauncher config (IPython.parallel.factory.RegistrationFactory attribute), 694 attribute), 824
config (IPython.parallel.apps.launcher.SSHControllerLauncher config_file_name (IPython.core.application.BaseIPythonApplication attribute), 697 attribute), 288
config (IPython.parallel.apps.launcher.SSHEngineLauncher config_file_name (IPython.core.profileapp.ProfileCreate attribute), 700 attribute), 517
config (IPython.parallel.apps.launcher.SSHEngineSetLauncher config_file_name (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 703 attribute), 588
config (IPython.parallel.apps.launcher.SSHLauncher config_file_name (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 706 attribute), 598
config (IPython.parallel.apps.launcher.WindowsHPCConfigFileLauncher (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 709 attribute), 604
config (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 712 attribute), 610
config (IPython.parallel.apps.launcher.WindowsHPCLauncher config_file_name (IPython.parallel.apps.ipcontrollerapp.IPController attribute), 715 attribute), 616
config (IPython.parallel.apps.logwatcher.LogWatcher config_file_name (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 718 attribute), 623
config (IPython.parallel.apps.winhpcjob.IPClusterJob config_file_name (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 723 attribute), 631
config (IPython.parallel.apps.winhpcjob.IPClusterTask config_file_paths (IPython.core.application.BaseIPythonApplication attribute), 725 attribute), 288
config (IPython.parallel.apps.winhpcjob.IPEngineSetJob config_file_paths (IPython.core.profileapp.ProfileCreate attribute), 728 attribute), 517
config (IPython.parallel.apps.winhpcjob.IPEngineTask config_file_paths (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 731 attribute), 588
config (IPython.parallel.apps.winhpcjob.WinHPCJob config_file_paths (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 734 attribute), 598
config (IPython.parallel.apps.winhpcjob.WinHPCTask config_file_paths (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 737 attribute), 604
config (IPython.parallel.controller.dictdb.BaseDB config_file_paths (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 777 attribute), 610
config (IPython.parallel.controller.dictdb.DictDB at- config_file_paths (IPython.parallel.apps.ipcontrollerapp.IPController attribute), 779 attribute), 616
config (IPython.parallel.controller.heartmonitor.HeartMonitor config_file_paths (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 782 attribute), 623
config (IPython.parallel.controller.hub.Hub at- config_file_paths (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 787 attribute), 631
config (IPython.parallel.controller.hub.HubFactory config_file_specified

(IPython.core.application.BaseIPythonApplication)
Configurable (class in IPython.config.configurable),
attribute), 288
config_file_specified
(IPython.core.profileapp.ProfileCreate
attribute), 517
config_file_specified
(IPython.parallel.apps.baseapp.BaseParallelApplication)
attribute), 588
config_file_specified
(IPython.parallel.apps.ipclusterapp.IPClusterEngines
attribute), 598
config_file_specified
(IPython.parallel.apps.ipclusterapp.IPClusterStart
attribute), 604
config_file_specified
(IPython.parallel.apps.ipclusterapp.IPClusterStop)
attribute), 610
config_file_specified
(IPython.parallel.apps.ipcontrollerapp.IPControllerApp
attribute), 616
config_file_specified
(IPython.parallel.apps.ipengineapp.IPEngineApp
attribute), 623
config_file_specified
(IPython.parallel.apps.iploggerapp.IPLLoggerApp)
attribute), 631
config_files (IPython.core.application.BaseIPythonApplication)
attribute), 288
config_files (IPython.core.profileapp.ProfileCreate
attribute), 517
config_files (IPython.parallel.apps.baseapp.BaseParallelApplication)
attribute), 588
config_files (IPython.parallel.apps.ipclusterapp.IPClusterEngines)
attribute), 599
config_files (IPython.parallel.apps.ipclusterapp.IPClusterStart)
attribute), 604
config_files (IPython.parallel.apps.ipclusterapp.IPClusterStop)
attribute), 610
config_files (IPython.parallel.apps.ipcontrollerapp.IPControllerApp)
attribute), 616
config_files (IPython.parallel.apps.ipengineapp.IPEngineApp)
attribute), 623
config_files (IPython.parallel.apps.iploggerapp.IPLLoggerApp)
attribute), 631
ConfigError (class in IPython.config.loader), 280
ConfigLoader (class in IPython.config.loader), 280
ConfigLoaderError (class in IPython.config.loader),
281
Configurable (class in IPython.config.configurable),
268
ConfigurableError (class in IPython.config.configurable), 270
connect_engine_logger() (in module IPython.parallel.util), 828
connection_request()
(IPython.parallel.controller.hub.Hub
method), 787
ConnectionError (class in IPython.parallel.error),
814
construct() (IPython.parallel.controller.hub.HubFactory
method), 792
Container (class in IPython.utils.traits), 914
ControllerApp (IPython.parallel.apps.launcher.BatchSystemLauncher
attribute), 640
context (IPython.parallel.apps.launcher.LSFControllerLauncher
attribute), 646
context (IPython.parallel.apps.launcher.LSFEngineSetLauncher
attribute), 650
ContextParser (IPython.parallel.apps.launcher.LSFLauncher
attribute), 653
context (IPython.parallel.apps.launcher.PBSControllerLauncher
attribute), 677
context (IPython.parallel.apps.launcher.PBSEngineSetLauncher
attribute), 680
context (IPython.parallel.apps.launcher.PBSLauncher
attribute), 684
ContextParser (IPython.parallel.apps.launcher.SGEControllerLauncher
attribute), 687
context (IPython.parallel.apps.launcher.SGEEngineSetLauncher
attribute), 691
context (IPython.parallel.apps.launcher.SGELauncher
attribute), 694
ControllerApp (IPython.parallel.apps.logwatcher.LogWatcher
attribute), 718
context (IPython.parallel.controller.hub.Hub
attribute), 787
context (IPython.parallel.controller.hub.HubFactory
attribute), 792
context (IPython.parallel.controller.scheduler.TaskScheduler
attribute), 796
context (IPython.parallel.engine.EngineFactory
attribute), 804

context (IPython.parallel.engine.streamkernel.Kernel
attribute), 809

context (IPython.parallel.factory.RegistrationFactory
attribute), 824

context() (IPython.core.ultratb.AutoFormattedTB
method), 537

context() (IPython.core.ultratb.ColorTB method),
539

context() (IPython.core.ultratb.FormattedTB
method), 540

control (IPython.parallel.controller.hub.EngineConnect
attribute), 785

control (IPython.parallel.controller.hub.HubFactory
attribute), 792

control_handlers (IPython.parallel.engine.streamkernel.~~K~~
attribute), 809

control_stream (IPython.parallel.engine.streamkernel.~~K~~
attribute), 809

controller_args (IPython.parallel.apps.launcher.LocalController.~~L~~
attribute), 657

controller_args (IPython.parallel.apps.launcher.MPIExecutor.~~E~~
attribute), 668

controller_args (IPython.parallel.apps.winhpcjob.IPController.~~C~~
attribute), 725

controller_cmd (IPython.parallel.apps.launcher.LocalController.~~L~~
attribute), 657

controller_cmd (IPython.parallel.apps.launcher.MPIExecutor.~~E~~
attribute), 668

controller_cmd (IPython.parallel.apps.winhpcjob.IPController.~~C~~
attribute), 725

controller_launcher_class
(IPython.parallel.apps.ipclusterapp.IPClusterStart
attribute), 604

ControllerCreationError (class in IPython.parallel.error), 814

ControllerError (class in IPython.parallel.error), 814

convert_field() (IPython.utils.text.EvalFormatter
method), 890

copy (IPython.parallel.client.client.Metadata
attribute), 751

copy (IPython.parallel.controller.dependency.Dependency
attribute), 773

copy (IPython.parallel.util.Namespace attribute),
826

copy (IPython.parallel.util.ReverseDict attribute),
827

copy (IPython.testing.globalipapp.ipnsdict
attribute), 833

copy() (IPython.config.loader.Config method), 279

copy() (IPython.utils.coloransi.ColorScheme
method), 855

copy() (IPython.utils.coloransi.ColorSchemeTable
method), 855

copy() (IPython.utils.ipstruct.Struct method), 868

copy_config_file() (IPython.core.profiledir.ProfileDir
method), 527

copy_config_files (IPython.core.application.BaseIPythonApplication
attribute), 288

copy_config_files (IPython.core.profileapp.ProfileCreate
attribute), 517

copy_config_files (IPython.parallel.apps.baseapp.BaseParallelApplic
attribute), 588

copy_config_files (IPython.parallel.apps.ipclusterapp.IPClusterEngin
attribute), 599

copy_config_files (IPython.parallel.apps.ipclusterapp.IPClusterStart
attribute), 604

copy_controller_file() (IPython.parallel.apps.ipclusterapp.IPClusterStop
attribute), 610

copy_controller_file() (IPython.parallel.apps.ipcontrollerapp.IPController
attribute), 616

copy_controller_file (IPython.parallel.apps.ipengineapp.IPEngineApp
attribute), 623

copy_controller_file (IPython.parallel.apps.iploggerapp.IPLLoggerApp
attribute), 631

count (IPython.utils.text.SList attribute), 895

ControllerFileNames() (in module
IPython.testing.ipunitest), 838

crash_handler_class
(IPython.core.application.BaseIPythonApplication
attribute), 288

crash_handler_class
(IPython.core.profileapp.ProfileCreate
attribute), 518

crash_handler_class
(IPython.parallel.apps.baseapp.BaseParallelApplication
attribute), 588

crash_handler_class
(IPython.parallel.apps.ipclusterapp.IPClusterEngines
attribute), 599

crash_handler_class
(IPython.parallel.apps.ipclusterapp.IPClusterStart
attribute), 604

crash_handler_class
(IPython.parallel.apps.ipclusterapp.IPClusterStop
attribute), 610

crash_handler_class
 (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 616
crash_handler_class
 (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623
crash_handler_class
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 631
CrashHandler (class in IPython.core.crashhandler), 304
create_profile_dir() (IPython.core.profiledir.ProfileDir class method), 527
create_profile_dir_by_name()
 (IPython.core.profiledir.ProfileDir method), 527
create_typestr2type_dicts() (in module IPython.utils.wildcard), 936
created (IPython.config.application.Application attribute), 264
created (IPython.config.configurable.Configurable attribute), 269
created (IPython.config.configurable.LoggingConfigurable attribute), 272
created (IPython.config.configurable.SingletonConfigurable attribute), 274
created (IPython.core.alias.AliasManager attribute), 285
created (IPython.core.application.BaseIPythonApplication attribute), 288
created (IPython.core.builtin_trap.BuiltinTrap attribute), 295
created (IPython.core.display_trap.DisplayTrap attribute), 317
created (IPython.core.displayhook.DisplayHook attribute), 320
created (IPython.core.displaypub.DisplayPublisher attribute), 323
created (IPython.core.extensions.ExtensionManager attribute), 329
created (IPython.core.formatters.BaseFormatter attribute), 332
created (IPython.core.formatters.DisplayFormatter attribute), 335
created (IPython.core.formatters.HTMLFormatter attribute), 338
created (IPython.core.formatters.JavascriptFormatter attribute), 344
 created (IPython.core.formatters.JSONFormatter attribute), 341
 created (IPython.core.formatters.LatexFormatter attribute), 346
 created (IPython.core.formatters.PlainTextFormatter attribute), 352
 created (IPython.core.formatters.PNGFormatter attribute), 349
 created (IPython.core.formatters.SVGFormatter attribute), 356
 created (IPython.core.history.HistoryManager attribute), 359
 created (IPython.core.interactiveshell.InteractiveShell attribute), 379
 created (IPython.core.payload.PayloadManager attribute), 455
 created (IPython.core.plugin.Plugin attribute), 458
 created (IPython.core.plugin.PluginManager attribute), 459
 created (IPython.core.prefilter.AliasChecker attribute), 462
 created (IPython.core.prefilter.AliasHandler attribute), 464
 created (IPython.core.prefilter.AssignMagicTransformer attribute), 466
 created (IPython.core.prefilter.AssignmentChecker attribute), 470
 created (IPython.core.prefilter.AssignSystemTransformer attribute), 468
 created (IPython.core.prefilter.AutocallChecker attribute), 475
 created (IPython.core.prefilter.AutoHandler attribute), 471
 created (IPython.core.prefilter.AutoMagicChecker attribute), 473
 created (IPython.core.prefilter.EmacsChecker attribute), 477
 created (IPython.core.prefilter.EmacsHandler attribute), 479
 created (IPython.core.prefilter.EscCharsChecker attribute), 481
 created (IPython.core.prefilter.HelpHandler attribute), 482
 created (IPython.core.prefilter.IPyAutocallChecker attribute), 484
 created (IPython.core.prefilter.IPyPromptTransformer attribute), 486
 created (IPython.core.prefilter.MacroChecker

attribute), 489
created (IPython.core.prefilter.MacroHandler attribute), 490
created (IPython.core.prefilter.MagicHandler attribute), 492
created (IPython.core.prefilter.MultilineMagicChecker attribute), 494
created (IPython.core.prefilter.PrefilterChecker attribute), 496
created (IPython.core.prefilter.PrefilterHandler attribute), 498
created (IPython.core.prefilter.PrefilterManager attribute), 500
created (IPython.core.prefilter.PrefilterTransformer attribute), 503
created (IPython.core.prefilter.PyPromptTransformer attribute), 505
created (IPython.core.prefilter.PythonOpsChecker attribute), 507
created (IPython.core.prefilter.ShellEscapeChecker attribute), 509
created (IPython.core.prefilter.ShellEscapeHandler attribute), 511
created (IPython.core.profileapp.ProfileApp attribute), 513
created (IPython.core.profileapp.ProfileCreate attribute), 518
created (IPython.core.profileapp.ProfileList attribute), 522
created (IPython.core.profiledir.ProfileDir attribute), 527
created (IPython.core.shellapp.InteractiveShellApp attribute), 534
created (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 588
created (IPython.parallel.apps.ipclusterapp.IPClusterApplication attribute), 594
created (IPython.parallel.apps.ipclusterapp.IPClusterEngineAttribute), 599
created (IPython.parallel.apps.ipclusterapp.IPClusterStartAttribute), 605
created (IPython.parallel.apps.ipclusterapp.IPClusterStopAttribute), 610
created (IPython.parallel.apps.ipcontrollerapp.IPControllerApplication attribute), 616
created (IPython.parallel.apps.ipengineapp.IPEngineApplication attribute), 623
created (IPython.parallel.apps.ipengineapp.MPI attribute), 628
created (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 631
created (IPython.parallel.apps.launcher.BaseLauncher attribute), 637
created (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 640
created (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 643
created (IPython.parallel.apps.launcher.LocalControllerLauncher attribute), 657
created (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 659
created (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 662
created (IPython.parallel.apps.launcher.LocalProcessLauncher attribute), 665
created (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 646
created (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 650
created (IPython.parallel.apps.launcher.LSFLauncher attribute), 653
created (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 668
created (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 671
created (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 674
created (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 677
created (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 680
created (IPython.parallel.apps.launcher.PBSLauncher attribute), 684
created (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 687
created (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 691
created (IPython.parallel.apps.launcher.SGELauncher attribute), 694
created (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 697
created (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 700
created (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 703
created (IPython.parallel.apps.launcher.SSHLauncher attribute), 704

attribute), 706
 created (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher
 attribute), 709
 cwd_filt() (IPython.core.prompts.Prompt1 method),
 created (IPython.parallel.apps.launcher.WindowsHPCControllerSetLauncher
 attribute), 712
 cwd_filt() (IPython.core.prompts.Prompt2 method),
 created (IPython.parallel.apps.launcher.WindowsHPCControllerSetLauncher
 attribute), 715
 cwd_filt() (IPython.core.prompts.PromptOut
 method), 531
 cwd_filt2() (IPython.core.prompts.BasePrompt
 method), 530
 created (IPython.parallel.apps.logwatcher.LogWatcher
 attribute), 718
 cwd_filt2() (IPython.core.prompts.Prompt1
 created (IPython.parallel.apps.winhpcjob.IPControllerJob
 attribute), 723
 cwd_filt2() (IPython.core.prompts.Prompt2
 created (IPython.parallel.apps.winhpcjob.IPControllerTask
 attribute), 725
 cwd_filt2() (IPython.core.prompts.PromptOut
 created (IPython.parallel.apps.winhpcjob.IPEngineSetJob
 attribute), 728
 Cyan (IPython.utils.coloransi.InputTermColors
 created (IPython.parallel.apps.winhpcjob.IPEngineTask
 attribute), 731
 Cyan (IPython.utils.coloransi.TermColors attribute),
 created (IPython.parallel.apps.winhpcjob.WinHPCJob
 attribute), 734
 created (IPython.parallel.apps.winhpcjob.WinHPCTaskD
 attribute), 737
 created (IPython.parallel.controller.dictdb.BaseDB
 attribute), 777
 created (IPython.parallel.controller.dictdb.DictDB
 attribute), 779
 created (IPython.parallel.controller.heartmonitor.HeartMonitor
 attribute), 782
 created (IPython.parallel.controller.hub.Hub
 attribute), 788
 created (IPython.parallel.controller.hub.HubFactory
 attribute), 792
 created (IPython.parallel.controller.scheduler.TaskScheduler
 attribute), 796
 created (IPython.parallel.controller.sqlitedb.SQLiteDB
 attribute), 801
 created (IPython.parallel.engine.engine.EngineFactory
 attribute), 804
 created (IPython.parallel.engine.streamkernel.Kernel
 attribute), 810
 created (IPython.parallel.factory.RegistrationFactory
 attribute), 824
 CUnicode (class in IPython.utils.traits), 910
 current_gui() (IPython.lib.inpthook.InputHookManager
 method), 570
 current_length() (IPython.core.interactiveshell.ReadLineNoRecog
 method), 415
 cwd_filt() (IPython.core.prompts.BasePrompt
 method), 530
 daemon (IPython.core.history.HistorySavingThread
 attribute), 363
 daemon (IPython.lib.backgroundjobs.BackgroundJobBase
 attribute), 548
 daemon (IPython.lib.backgroundjobs.BackgroundJobExpr
 attribute), 549
 daemon (IPython.lib.backgroundjobs.BackgroundJobFunc
 attribute), 550
 daemon (IPython.parallel.apps.win32support.ForwarderThread
 attribute), 720
 daemonize (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 attribute), 599
 daemonize (IPython.parallel.apps.ipclusterapp.IPClusterStart
 attribute), 605
 daemonize() (in module IPython.utils.daemonize),
 859
 DarkGray (IPython.utils.coloransi.InputTermColors
 attribute), 857
 DarkGray (IPython.utils.coloransi.TermColors
 attribute), 858
 date_default() (in module IPython.utils.jsonutil), 871
 db (IPython.core.history.HistoryManager attribute),
 359
 db (IPython.parallel.controller.hub.Hub
 attribute), 788
 db (IPython.parallel.controller.hub.HubFactory
 attribute), 792

db_cache_size (IPython.core.history.HistoryManager attribute), 359
db_class (IPython.parallel.controller.hub.HubFactory attribute), 792
db_input_cache (IPython.core.history.HistoryManager attribute), 359
db_log_output (IPython.core.history.HistoryManager attribute), 360
db_output_cache (IPython.core.history.HistoryManager attribute), 360
db_query() (IPython.parallel.client.client.Client method), 747
db_query() (IPython.parallel.controller.hub.Hub method), 788
deactivate() (IPython.core.builtin_trap.BuiltinTrap method), 295
dead_engines (IPython.parallel.controller.hub.Hub attribute), 788
debug (IPython.core.interactiveshell.InteractiveShell attribute), 379
debug (IPython.parallel.client.client.Client attribute), 747
debugger() (IPython.core.interactiveshell.InteractiveShell method), 379
debugger() (IPython.core.ultratb.AutoFormattedTB method), 537
debugger() (IPython.core.ultratb.ColorTB method), 539
debugger() (IPython.core.ultratb.FormattedTB method), 541
debugger() (IPython.core.ultratb.VerboseTB method), 546
debugx() (in module IPython.utils.frame), 861
decode (IPython.utils.text.LSString attribute), 891
decorate_fn_with_doc() (in module IPython.core.debugger), 314
dedent() (in module IPython.utils.text), 897
deep_import_hook() (in module IPython.lib.deepreload), 553
deep_reload (IPython.core.interactiveshell.InteractiveShell attribute), 379
deep_reload_hook() (in module IPython.lib.deepreload), 553
default() (IPython.core.debugger.Pdb method), 307
default_aliases (IPython.core.alias.AliasManager attribute), 285
default_aliases() (in module IPython.core.alias), 287
default_argv() (in module IPython.testing.tools), 846
default_config() (in module IPython.testing.tools), 846
default_inits (IPython.parallel.apps.ipengineapp.MPI attribute), 628
default_log_level (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 599
default_log_level (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 605
default_option() (IPython.core.interactiveshell.InteractiveShell method), 379
default_option() (IPython.core.magic.Magic method), 421
default_template (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 640
default_template (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 646
default_template (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 650
default_template (IPython.parallel.apps.launcher.LSFLauncher attribute), 653
default_template (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 677
default_template (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 680
default_template (IPython.parallel.apps.launcher.PBSLauncher attribute), 684
default_template (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 687
default_template (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 691
default_template (IPython.parallel.apps.launcher.SGELauncher attribute), 694
default_value (IPython.core.interactiveshell.SeparateUnicode attribute), 416
default_value (IPython.utils.traits.Any attribute), 901
default_value (IPython.utils.traits.Bool attribute), 902
default_value (IPython.utils.traits.Bytes attribute), 903
default_value (IPython.utils.traits.CaselessStrEnum attribute), 911
default_value (IPython.utils.traits.CBool attribute), 904
default_value (IPython.utils.traits.CBytes attribute), 905
default_value (IPython.utils.traits.CComplex attribute), 906

| | | | | |
|---|-----|-----|--|-----|
| default_value (IPython.utils.traits.CFloat attribute), | 907 | at- | DefaultValueGenerator (class in IPython.utils.traits), | 915 |
| default_value (IPython.utils.traits.CInt attribute), | 908 | at- | deferred_printers (IPython.core.formatters.BaseFormatter attribute), | 332 |
| default_value (IPython.utils.traits.ClassBasedTraitType attribute), | 912 | at- | deferred_printers (IPython.core.formatters.HTMLFormatter attribute), | 338 |
| default_value (IPython.utils.traits.CLong attribute), | 909 | at- | deferred_printers (IPython.core.formatters.JavascriptFormatter attribute), | 344 |
| default_value (IPython.utils.traits.Complex attribute), | 913 | at- | deferred_printers (IPython.core.formatters.JSONFormatter attribute), | 341 |
| default_value (IPython.utils.traits.Container attribute), | 914 | at- | deferred_printers (IPython.core.formatters.LatexFormatter attribute), | 346 |
| default_value (IPython.utils.traits.CUnicode attribute), | 910 | at- | deferred_printers (IPython.core.formatters.PlainTextFormatter attribute), | 352 |
| default_value (IPython.utils.traits.Dict attribute), | 915 | at- | deferred_printers (IPython.core.formatters.PNGFormatter attribute), | 349 |
| default_value (IPython.utils.traits.DottedObjectName attribute), | 916 | at- | deferred_printers (IPython.core.formatters.SVGFormatter attribute), | 356 |
| default_value (IPython.utils.traits.Enum attribute), | 917 | at- | define_alias() (IPython.core.alias.AliasManager method), | 285 |
| default_value (IPython.utils.traits.Float attribute), | 918 | at- | define_macro() (IPython.core.interactiveshell.InteractiveShell method), | 379 |
| default_value (IPython.utils.traits.Instance attribute), | 920 | at- | define_magic() (IPython.core.interactiveshell.InteractiveShell method), | 379 |
| default_value (IPython.utils.traits.Int attribute), | 921 | at- | del_var() (IPython.core.interactiveshell.InteractiveShell method), | 379 |
| default_value (IPython.utils.traits.List attribute), | 923 | at- | delay (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), | 605 |
| default_value (IPython.utils.traits.Long attribute), | 924 | at- | delete_command (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), | 640 |
| default_value (IPython.utils.traits.ObjectName attribute), | 925 | at- | delete_command (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), | 646 |
| default_value (IPython.utils.traits.Set attribute), | 927 | at- | delete_command (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), | 650 |
| default_value (IPython.utils.traits.TCPAddress attribute), | 928 | at- | delete_command (IPython.parallel.apps.launcher.LSFLauncher attribute), | 653 |
| default_value (IPython.utils.traits.This attribute), | 929 | at- | delete_command (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), | 677 |
| default_value (IPython.utils.traits.TraitType attribute), | 930 | at- | delete_command (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), | 680 |
| default_value (IPython.utils.traits.Tuple attribute), | 931 | at- | delete_command (IPython.parallel.apps.launcher.PBSLauncher attribute), | 684 |
| default_value (IPython.utils.traits.Type attribute), | 933 | at- | delete_command (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), | 687 |
| default_value (IPython.utils.traits.Unicode attribute), | 933 | at- | delete_command (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), | 691 |
| defaultFile() (IPython.core.debugger.Pdb method), | 307 | at- | delete_command (IPython.parallel.apps.launcher.SGELauncher attribute), | 694 |

Demo (class in IPython.lib.demo), 561
DemoError (class in IPython.lib.demo), 562
depend (class in IPython.parallel.controller.dependency), 774
Dependency (class in IPython.parallel.controller.dependency), 772
DependencyTimeout (class in IPython.parallel.error), 814
dependent (class in IPython.parallel.controller.dependency), 775
depending (IPython.parallel.controller.scheduler.TaskScheduler), 796
deq() (IPython.lib.pretty.GroupQueue method), 582
description (IPython.config.application.Application attribute), 264
description (IPython.core.application.BaseIPythonApplication attribute), 288
description (IPython.core.profileapp.ProfileApp attribute), 513
description (IPython.core.profileapp.ProfileCreate attribute), 518
description (IPython.core.profileapp.ProfileList attribute), 522
description (IPython.parallel.apps.baseapp.BaseParallelApp attribute), 588
description (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 594
description (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 599
description (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 605
description (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 610
description (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 616
description (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623
description (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 631
destinations (IPython.parallel.controller.scheduler.TaskScheduler), 796
determine_parent() (in IPython.lib.deepreload), 554
device (IPython.parallel.controller.heartmonitor.Heart attribute), 781
grep() (in module IPython.utils.text), 897

dhook_wrap() (in module IPython.utils.doctestreload), 861
Dict (class in IPython.utils.traits), 915
dict() (IPython.utils.ipstruct.Struct method), 868
dict_dir() (in module IPython.utils.wildcard), 936
DictDB (class in IPython.parallel.controller.dictdb), 778
difference (IPython.parallel.controller.dependency.Dependency attribute), 773
difference_update (IPython.parallel.controller.dependency.Dependency attribute), 773
dir2() (in module IPython.utils.dir2), 860
HistoryManager (IPython.core.history.HistoryManager attribute), 360
direct_view() (IPython.parallel.client.client.Client method), 747
DirectView (class in IPython.parallel.client.view), 756
disable_gtk() (IPython.lib.inupthook.InputHookManager method), 570
disable_qt4() (IPython.lib.inupthook.InputHookManager method), 571
disable_tk() (IPython.lib.inupthook.InputHookManager method), 571
disable_wx() (IPython.lib.inupthook.InputHookManager method), 571
disambiguate_ip_address() (in module IPython.parallel.util), 828
disambiguate_url() (in module IPython.parallel.util), 828
discard (IPython.parallel.controller.dependency.Dependency attribute), 773
dispatch() (IPython.utils.strdispatch.StrDispatch method), 886
dispatch_call() (IPython.core.debugger.Pdb method), 307
dispatch_control() (IPython.parallel.engine.streamkernel.Kernel method), 307
dispatch_custom_completer() (IPython.core.completer.IPCompleter method), 301
dispatch_db() (IPython.parallel.controller.hub.Hub method), 788
dispatch_exception() (IPython.core.debugger.Pdb method), 307
dispatch_line() (IPython.core.debugger.Pdb method), 307
dispatch_monitor_traffic()

(IPython.parallel.controller.hub.Hub
method), 788
dispatch_notification()
(IPython.parallel.controller.scheduler.TaskScheduler
method), 796
dispatch_query() (IPython.parallel.controller.hub.Hub
method), 788
dispatch_queue() (IPython.parallel.engine.streamkernel.Kernel
method), 810
dispatch_reply() (IPython.parallel.engine.kernelstarter.KernelStarter
method), 806
dispatch_request() (IPython.parallel.engine.kernelstarter.KernelStarter
method), 806
dispatch_result() (IPython.parallel.controller.scheduler.TaskScheduler
method), 796
dispatch_return()
(IPython.core.debugger.Pdb
method), 307
dispatch_submission()
(IPython.parallel.controller.scheduler.TaskScheduler
method), 796
display() (in module IPython.core.display), 315
display_formatter (IPython.core.interactiveshell.InteractiveShell
attribute), 380
display_hook_factory
(IPython.parallel.engine.EngineFactory
attribute), 804
display_html() (in module IPython.core.display),
315
display_javascript()
(in module IPython.core.display), 315
display_json() (in module IPython.core.display), 315
display_latex() (in module IPython.core.display),
315
display_png() (in module IPython.core.display), 316
display_pretty() (in module IPython.core.display),
316
display_pub_class (IPython.core.interactiveshell.InteractiveShell
attribute), 380
display_svg() (in module IPython.core.display), 316
display_trap (IPython.core.interactiveshell.InteractiveShell
attribute), 380
DisplayFormatter (class in IPython.core.formatters),
334
DisplayHook (class in IPython.core.displayhook),
319
displayhook() (IPython.core.debugger.Pdb
method), 307
displayhook_class (IPython.core.interactiveshell.InteractiveShell
attribute), 380
DisplayPublisher (class in IPython.core.displaypub),
322
DisplayTrap (class in IPython.core.display_trap),
316
do_a() (IPython.core.debugger.Pdb
method), 307
do_alias() (IPython.core.debugger.Pdb
method), 307
do_b() (IPython.core.debugger.Pdb
method), 308
do_commands()
(IPython.core.debugger.Pdb
method), 308
do_condition()
(IPython.core.debugger.Pdb
method), 308
do_continue() (IPython.core.debugger.Pdb
method),
308
do_d() (IPython.core.debugger.Pdb
method), 308
do_debug()
(IPython.core.debugger.Pdb
method),
308
do_disable()
(IPython.core.debugger.Pdb
method),
308
do_down()
(IPython.core.debugger.Pdb
method),
308
do_enable()
(IPython.core.debugger.Pdb
method),
308
do_EOF()
(IPython.core.debugger.Pdb
method), 307
do_exit()
(IPython.core.debugger.Pdb
method), 308
do_h()
(IPython.core.debugger.Pdb
method), 308
do_help()
(IPython.core.debugger.Pdb
method), 308
do_ignore()
(IPython.core.debugger.Pdb
method),
308
do_import_statements()
(IPython.parallel.apps.ipcontrollerapp.IPCControllerApp
method), 616
do_j()
(IPython.core.debugger.Pdb
method), 308
do_jump()
(IPython.core.debugger.Pdb
method),
308
do_l()
(IPython.core.debugger.Pdb
method), 308
do_list()
(IPython.core.debugger.Pdb
method), 308
do_n()
(IPython.core.debugger.Pdb
method), 308
do_next()
(IPython.core.debugger.Pdb
method), 308
do_reload()
(IPython.core.debugger.Pdb
method), 309

do_pdef() (IPython.core.debugger.Pdb method), 309
do_pdoc() (IPython.core.debugger.Pdb method), 309
do_pinfo() (IPython.core.debugger.Pdb method), 309
do_pp() (IPython.core.debugger.Pdb method), 309
do_q() (IPython.core.debugger.Pdb method), 309
do_quit() (IPython.core.debugger.Pdb method), 309
do_r() (IPython.core.debugger.Pdb method), 309
do_restart() (IPython.core.debugger.Pdb method), 309
do_return() (IPython.core.debugger.Pdb method), 309
do_retval() (IPython.core.debugger.Pdb method), 309
do_run() (IPython.core.debugger.Pdb method), 309
do(rv) (IPython.core.debugger.Pdb method), 309
do_s() (IPython.core.debugger.Pdb method), 309
do_step() (IPython.core.debugger.Pdb method), 309
do_tbreak() (IPython.core.debugger.Pdb method), 309
do_u() (IPython.core.debugger.Pdb method), 309
do_unalias() (IPython.core.debugger.Pdb method), 309
do_unt() (IPython.core.debugger.Pdb method), 309
do_until() (IPython.core.debugger.Pdb method), 309
do_up() (IPython.core.debugger.Pdb method), 309
do_w() (IPython.core.debugger.Pdb method), 309
do_whatis() (IPython.core.debugger.Pdb method), 309
do_where() (IPython.core.debugger.Pdb method), 309
Doc2UnitTester (class in IPython.testing.ipunitest), 837
doc_header (IPython.core.debugger.Pdb attribute), 309
doc_leader (IPython.core.debugger.Pdb attribute), 309
doctest_ivars() (in module IPython.testing.plugin.test_refs), 844
doctest_multiline1() (in module IPython.testing.plugin.test_ipdoctest), 843
doctest_multiline2() (in module IPython.testing.plugin.test_ipdoctest), 844
doctest_multiline3() (in module IPython.testing.plugin.test_ipdoctest), 844
doctest_refs() (in module IPython.testing.plugin.test_refs), 844
doctest_reload() (in module IPython.utils.doctestreload), 861
doctest_run() (in module IPython.testing.plugin.test_refs), 844
doctest_runvars() (in module IPython.testing.plugin.test_refs), 845
doctest_simple() (in module IPython.testing.plugin.test_ipdoctest), 844
DottedObjectName (class in IPython.utils.traits), 916
drop_matching_records() (IPython.parallel.controller.dictdb.DictDB method), 779
drop_matching_records() (IPython.parallel.controller.sqlitedb.SQLiteDB method), 801
drop_record() (IPython.parallel.controller.dictdb.DictDB method), 779
drop_record() (IPython.parallel.controller.sqlitedb.SQLiteDB method), 801

E

edit() (IPython.lib.demo.ClearDemo method), 557
edit() (IPython.lib.demo.ClearIPDemo method), 559
edit() (IPython.lib.demo.Demo method), 561
edit() (IPython.lib.demo.IPythonDemo method), 563
edit() (IPython.lib.demo.IPythonLineDemo method), 565
edit() (IPython.lib.demo.LineDemo method), 567
EDITOR, 190
editor() (in module IPython.core.hooks), 367
element_error() (IPython.utils.traits.Container method), 914
element_error() (IPython.utils.traits.List method), 923
element_error() (IPython.utils.traits.Set method), 927
element_error() (IPython.utils.traits.Tuple method), 931
EmacsChecker (class in IPython.core.prefilter), 476
EmacsHandler (class in IPython.core.prefilter), 478
empty_record() (in module IPython.parallel.controller.hub), 794
emptyline() (IPython.core.debugger.Pdb method), 309

enable_gtk() (IPython.lib.inpthook.InputHookManager attribute), 486
 method), 571
enable_gui() (in module IPython.lib.inpthook), 572
enable_pylab() (IPython.core.interactiveshell.InteractiveShell attribute), 494
 method), 380
enable_qt4() (IPython.lib.inpthook.InputHookManager attribute), 496
 method), 571
enable_tk() (IPython.lib.inpthook.InputHookManager attribute), 503
 method), 571
enable_wx() (IPython.lib.inpthook.InputHookManager attribute), 505
 method), 572
enabled (IPython.core.formatters.BaseFormatter attribute), 332
enabled (IPython.core.formatters.FormatterABC attribute), 337
enabled (IPython.core.formatters.HTMLFormatter attribute), 338
enabled (IPython.core.formatters.JavascriptFormatter attribute), 344
enabled (IPython.core.formatters.JSONFormatter attribute), 341
enabled (IPython.core.formatters.LatexFormatter attribute), 347
enabled (IPython.core.formatters.PlainTextFormatter attribute), 352
enabled (IPython.core.formatters.PNGFormatter attribute), 349
enabled (IPython.core.formatters.SVGFormatter attribute), 356
enabled (IPython.core.prefilter.AliasChecker attribute), 462
enabled (IPython.core.prefilter.AssignMagicTransformer attribute), 466
enabled (IPython.core.prefilter.AssignmentChecker attribute), 470
enabled (IPython.core.prefilter.AssignSystemTransformer attribute), 468
enabled (IPython.core.prefilter.AutocallChecker attribute), 475
enabled (IPython.core.prefilter.AutoMagicChecker attribute), 473
enabled (IPython.core.prefilter.EmacsChecker attribute), 477
enabled (IPython.core.prefilter.EscCharsChecker attribute), 481
enabled (IPython.core.prefilter.IPyAutocallChecker attribute), 484
enabled (IPython.core.prefilter.IPyPromptTransformer attribute), 486
 method), 571
enabled (IPython.core.prefilter.MacroChecker attribute), 489
enabled (IPython.core.prefilter.MultiLineMagicChecker attribute), 494
enabled (IPython.core.prefilter.PrefilterChecker attribute), 496
enabled (IPython.core.prefilter.PrefilterTransformer attribute), 503
enabled (IPython.core.prefilter.PyPromptTransformer attribute), 505
enabled (IPython.core.prefilter.PythonOpsChecker attribute), 507
enabled (IPython.core.prefilter.ShellEscapeChecker attribute), 509
ename (IPython.parallel.error.CompositeError attribute), 813
ename (IPython.parallel.error.RemoteError attribute), 820
encode (IPython.utils.text.LSString attribute), 891
encoding (IPython.core.inputsplitter.InputSplitter attribute), 371
encoding (IPython.core.inputsplitter.IPythonInputSplitter attribute), 369
end_group() (IPython.lib.pretty.PrettyPrinter method), 583
end_group() (IPython.lib.pretty.RepresentationPrinter method), 584
end_session() (IPython.core.history.HistoryManager method), 360
endswith (IPython.utils.text.LSString attribute), 891
engine_args (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 659
engine_args (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 662
engine_args (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 703
engine_args (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 731
engine_cmd (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 660
engine_cmd (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 731
engine_info (IPython.parallel.controller.hub.Hub attribute), 788
engine_info (IPython.parallel.error.CompositeError attribute), 813
engine_info (IPython.parallel.error.RemoteError attribute), 820

tribute), 820
engine_ip (IPython.parallel.controller.hub.HubFactory attribute), 792
engine_launcher_class (IPython.parallel.apps.ipclusterapp.IPCluster attribute), 599
engine_launcher_class (IPython.parallel.apps.ipclusterapp.IPCluster attribute), 605
engine_stream (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
engine_transport (IPython.parallel.controller.hub.HubFactory attribute), 792
EngineConnector (class in IPython.parallel.controller.hub), 784
EngineCreationError (class in IPython.parallel.error), 815
EngineError (class in IPython.parallel.error), 815
EngineFactory (class in IPython.parallel.engine.engine), 803
engines (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 703
engines (IPython.parallel.controller.hub.Hub attribute), 788
enq() (IPython.lib.pretty.GroupQueue method), 582
ensure_fromlist() (in module IPython.lib.deepreload), 554
Enum (class in IPython.utils.traits), 917
environment variable
 %PATH%, 140
 EDITOR, 190
 PYTHON_DIR, 183
 PATH, 2
 PYTHONSTARTUP, 943
environment_variables
 (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 725
environment_variables
 (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 731
environment_variables
 (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 737
error() (in module IPython.utils.warn), 935
error() (IPython.config.loader.ArgumentParser method), 278
error() (IPython.core.interactiveshell.SeparateUnicode method), 416
error() (IPython.core.magic_arguments.MagicArgumentParser method), 446
error() (IPython.utils.traits.Any method), 901
error() (IPython.utils.traits.Bool method), 902
error() (IPython.utils.traits.Bytes method), 903
error() (IPython.utils.traits.CaselessStrEnum method), 911
error() (IPython.utils.traits.CBool method), 904
error() (IPython.utils.traits.CBytes method), 905
error() (IPython.utils.traits.CComplex method), 906
error() (IPython.utils.traits.CFloat method), 907
error() (IPython.utils.traits.CInt method), 908
error() (IPython.utils.traits.ClassBasedTraitType method), 912
error() (IPython.utils.traits.CLong method), 909
error() (IPython.utils.traits.Complex method), 913
error() (IPython.utils.traits.Container method), 914
error() (IPython.utils.traits.CUnicode method), 915
error() (IPython.utils.traits.Dict method), 915
error() (IPython.utils.traits.DottedObjectName method), 916
error() (IPython.utils.traits.Enum method), 917
error() (IPython.utils.traits.Float method), 918
error() (IPython.utils.traits.Instance method), 920
error() (IPython.utils.traits.Int method), 921
error() (IPython.utils.traits.List method), 923
error() (IPython.utils.traits.Long method), 924
error() (IPython.utils.traits.ObjectName method), 925
error() (IPython.utils.traits.Set method), 927
error() (IPython.utils.traits.TCPAddress method), 928
error() (IPython.utils.traits.This method), 929
error() (IPython.utils.traits.TraitType method), 930
error() (IPython.utils.traits.Tuple method), 931
error() (IPython.utils.traits.Type method), 933
error() (IPython.utils.traits.Unicode method), 934
esc_quotes() (in module IPython.utils.text), 897
 esc_strings (IPython.core.prefilter.AliasHandler attribute), 464
 esc_strings (IPython.core.prefilter.AutoHandler attribute), 471
 esc_strings (IPython.core.prefilter.EmacsHandler attribute), 479

esc_strings (IPython.core.prefilter.HelpHandler attribute), 482
esc_strings (IPython.core.prefilter.MacroHandler attribute), 490
esc_strings (IPython.core.prefilter.MagicHandler attribute), 492
esc_strings (IPython.core.prefilter.PrefilterHandler attribute), 498
esc_strings (IPython.core.prefilter.ShellEscapeHandler exec_files (IPython.core.shellapp.InteractiveShellApp attribute), 511
EscapedTransformer (class in IPython.core.inputsplitter), 369
EscCharsChecker (class in IPython.core.prefilter), 480
ev() (IPython.core.interactiveshell.InteractiveShell method), 380
EvalDict (class in IPython.utils.attic), 850
EvalFormatter (class in IPython.utils.text), 889
evaluate (IPython.parallel.error.CompositeError attribute), 813
evaluate (IPython.parallel.error.RemoteError attribute), 820
ex() (IPython.core.interactiveshell.InteractiveShell method), 380
examples (IPython.config.application.Application attribute), 264
examples (IPython.core.application.BaseIPythonApplication attribute), 289
examples (IPython.core.profileapp.ProfileApp attribute), 513
examples (IPython.core.profileapp.ProfileCreate attribute), 518
examples (IPython.core.profileapp.ProfileList attribute), 522
examples (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 588
examples (IPython.parallel.apps.ipclusterapp.IPClusterEngine attribute), 594
examples (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 605
examples (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 610
examples (IPython.parallel.apps.ipcontrollerapp.IPController attribute), 616
examples (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623
examples (IPython.parallel.apps.iploggerapp.IPLoggerApp attribute), 631
excepthook() (IPython.core.interactiveshell.InteractiveShell method), 380
exception_colors() (in module IPython.core.excolors), 327
exclude_aliases() (IPython.core.alias.AliasManager method), 285
exec_lines (IPython.core.shellapp.InteractiveShellApp attribute), 534
exec_lines (IPython.parallel.engine.streamkernel.Kernel attribute), 810
execRcLines() (IPython.core.debugger.Pdb method), 310
execute() (IPython.parallel.client.view.DirectView method), 758
execute_request() (IPython.parallel.engine.streamkernel.Kernel method), 810
execution_count (IPython.core.interactiveshell.InteractiveShell attribute), 380
exit() (IPython.config.application.Application method), 264
exit() (IPython.config.loader.ArgumentParser method), 278
exit() (IPython.core.application.BaseIPythonApplication method), 289
exit() (IPython.core.magic_arguments.MagicArgumentParser method), 446
exit() (IPython.core.profileapp.ProfileApp method), 513
exit() (IPython.core.profileapp.ProfileCreate method), 518
exit() (IPython.core.profileapp.ProfileList method), 522
exit() (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 589
exit() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 594
exit() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 599
exit() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 605
exit() (IPython.parallel.apps.ipcontrollerapp.IPController method), 610
exit() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 616

exit() (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623
 method), 623
exit() (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 631
 method), 631
exit_now (IPython.core.interactiveshell.InteractiveShell attribute), 709
 attribute), 380
ExitAutocall (class in IPython.core.autocall), 292
exiter (IPython.core.interactiveshell.InteractiveShell attribute), 380
expand_alias() (IPython.core.alias.AliasManager method), 285
expand_aliases() (IPython.core.alias.AliasManager method), 285
expand_path() (in module IPython.utils.path), 878
expand_user() (in module IPython.core.completer), 302
expandtabs (IPython.utils.text.LSString attribute), 891
extend (IPython.utils.text.SList attribute), 895
extension_manager (IPython.core.interactiveshell.InteractiveShels_above() (in module IPython.utils.frame), 862
attribute), 381
ExtensionManager (class in IPython.core.extensions), 328
extensions (IPython.core.shellapp.InteractiveShellApp fail_unreachable() (IPython.parallel.controller.scheduler.TaskScheduler attribute), 534
 method), 797
extra_args (IPython.config.application.Application attribute), 264
extra_args (IPython.core.application.BaseIPythonApplication attribute), 289
extra_args (IPython.core.profileapp.ProfileApp attribute), 514
extra_args (IPython.core.profileapp.ProfileCreate attribute), 518
extra_args (IPython.core.profileapp.ProfileList attribute), 522
extra_args (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 589
extra_args (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 595
extra_args (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 599
extra_args (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 605
extra_args (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 611
extra_args (IPython.parallel.apps.ipcontrollerapp.IPCControllerAttribute), 801
 attribute), 616
extra_args (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 813
 extra_args (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 631
 extra_args (IPython.parallel.apps.launcher.WindowsHPCControllerL
attribute), 709
 extra_args (IPython.parallel.apps.launcher.WindowsHPCEngineSetL
attribute), 712
extra_extension (IPython.core.shellapp.InteractiveShellApp attribute), 534
extract_dates() (in module IPython.utils.jsonutil), 871
extract_hist_ranges() (in module IPython.core.history), 364
extract_input_lines() (IPython.core.interactiveshell.InteractiveShell method), 381
extract_input_lines() (IPython.core.magic.Magic method), 421
extract_vars() (in module IPython.utils.frame), 862
extractvarsabove() (in module IPython.utils.frame), 862
F
fail_unreachable() (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
failed (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
failure (IPython.parallel.controller.dependency.Dependency attribute), 773
fatal() (in module IPython.utils.warn), 935
fields() (IPython.utils.text.SList method), 895
figsize() (in module IPython.lib.pylabtools), 585
file_matches() (IPython.core.completer.ICompleter method), 301
file_read() (in module IPython.utils.io), 866
file_to_run (IPython.core.shellapp.InteractiveShellApp attribute), 534
FileConfigLoader (class in IPython.config.loader), 281
 filefind() (in module IPython.utils.path), 878
 filestart() (in module IPython.utils.path), 878
 filename (IPython.core.interactiveshell.InteractiveShell attribute), 381
 filename (IPython.parallel.controller.sqlitedb.SQLiteDatabase attribute), 801
 FileTimeoutError (class in IPython.parallel.error), 815
extra_args (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 815

filter_ns() (in module IPython.utils.wildcard), 936
 find (IPython.utils.text.LSString attribute), 891
 find_args() (IPython.parallel.apps.launcher.BaseLauncher),
 method), 637
 find_args() (IPython.parallel.apps.launcher.BatchSystem),
 method), 640
 find_args() (IPython.parallel.apps.launcher.IPClusterLan-
 method), 643
 find_args() (IPython.parallel.apps.launcher.LocalControllerLan-
 method), 657
 find_args() (IPython.parallel.apps.launcher.LocalEngineLan-
 method), 660
 find_args() (IPython.parallel.apps.launcher.LocalEngineSetLan-
 method), 662
 find_args() (IPython.parallel.apps.launcher.LocalProcessLaunch-
 method), 665
 find_args() (IPython.parallel.apps.launcher.LSFControllerLan-
 method), 646
 find_args() (IPython.parallel.apps.launcher.LSFEngineSetLan-
 method), 650
 find_args() (IPython.parallel.apps.launcher.LSFLaunch-
 method), 653
 find_args() (IPython.parallel.apps.launcher.MPIExecConf-
 method), 668
 find_args() (IPython.parallel.apps.launcher.MPIExecEngi-
 method), 671
 find_args() (IPython.parallel.apps.launcher.MPIExecLa-
 method), 674
 find_args() (IPython.parallel.apps.launcher.PBSControllerLan-
 method), 677
 find_args() (IPython.parallel.apps.launcher.PBSEngineSetLan-
 method), 680
 find_args() (IPython.parallel.apps.launcher.PBSLaunch-
 method), 684
 find_args() (IPython.parallel.apps.launcher.SGEControllerLan-
 method), 688
 find_args() (IPython.parallel.apps.launcher.SGEEngineSetLan-
 method), 691
 find_args() (IPython.parallel.apps.launcher.SGELauncher),
 method), 694
 find_args() (IPython.parallel.apps.launcher.SSHControllerLan-
 method), 697
 find_args() (IPython.parallel.apps.launcher.SSHEngineLan-
 method), 700
 find_args() (IPython.parallel.apps.launcher.SSHEngineSetLan-
 method), 704
 find_args() (IPython.parallel.apps.launcher.SSHLauncher),
 method), 706

find_args() (IPython.parallel.apps.launcher.WindowsHPCControllerLan-
 method), 709
 find_args() (IPython.parallel.apps.launcher.WindowsHPCEngineSetLan-
 method), 712
 find_args() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 715
 find_args() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 715
 find_args() (IPython.utils.process), 884
 method)
 find_args() (IPython.core.prefilter.PrefilterManager),
 method), 585
 find_args() (IPython.core.profiledir.ProfileDir),
 method), 527
 find_args() (IPython.core.profiledir.ProfileDir),
 method), 527
 find_records() (IPython.parallel.controller.dictdb.DictDB),
 method), 779
 find_head_package() (in module IPython.core.profiledir.ProfileDir),
 method), 500
 find_job_cmd() (in module IPython.core.profiledir.ProfileDir),
 method), 554
 find_profile_dir() (IPython.core.profiledir.ProfileDir),
 method), 717
 find_profile_dir_by_name() (IPython.core.profiledir.ProfileDir),
 method), 527
 find_profile_dir_by_name() (IPython.core.profiledir.ProfileDir),
 method), 527
 find_records() (IPython.parallel.controller.dictdb.DictDB),
 method), 801
 find_records() (IPython.parallel.apps.ipengineapp.IPEngineApp),
 method), 623
 find_user_code() (IPython.core.interactiveshell.InteractiveShell),
 method), 381
 find_usecheme() (IPython.parallel.apps.winhpcjob),
 method), 739
 SetHandlerError (class in IPython.utils.process), 884
 method), 884
 SetHandlerError (IPython.core.ultratb), 547
 findsource() (in module IPython.core.ultratb), 547
 finish_displayhook() (IPython.core.displayhook.DisplayHook),
 method), 684
 finish_job() (IPython.parallel.controller.scheduler.TaskScheduler),
 method), 320
 finish_registration() (IPython.parallel.controller.hub.Hub),
 method), 797
 fix_error_editor() (in module IPython.core.hooks),
 method), 788
 fix_frame_records_filenames() (in module IPython.core.ultratb),
 method), 167
 flag_calls() (in module IPython.utils.decorators),
 method), 547
 flags (IPython.config.application.Application),
 method), 160
 flags (IPython.core.application.BaseIPythonApplication),
 method), 264

attribute), 289
flags (IPython.core.profileapp.ProfileApp attribute), 514
flags (IPython.core.profileapp.ProfileCreate attribute), 518
flags (IPython.core.profileapp.ProfileList attribute), 522
flags (IPython.parallel.apps.baseapp.BaseParallelApp attribute), 589
flags (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 595
flags (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 599
flags (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 605
flags (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 611
flags (IPython.parallel.apps.ipcontrollerapp.IPController attribute), 617
flags (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 623
flags (IPython.parallel.apps.iploggerapp.IPLoaderApp attribute), 631
flags (IPython.parallel.client.remotefunction.ParallelFunction attribute), 755
flags (IPython.parallel.client.remotefunction.RemoteFuture attribute), 755
flat_matches() (IPython.utils.strdispatch.StrDispatch method), 886
flatten() (in module IPython.utils.data), 859
flatten_array() (IPython.parallel.client.map.RoundRobinMap method), 753
flatten_list() (IPython.parallel.client.map.RoundRobinMap method), 753
float() (IPython.lib.demo.ClearDemo method), 558
float() (IPython.lib.demo.ClearIPDemo method), 559
float() (IPython.lib.demo.Demo method), 562
float() (IPython.lib.demo.IPythonDemo method), 564
float() (IPython.lib.demo.IPythonLineDemo method), 565
float() (IPython.lib.demo.LineDemo method), 567
Float (class in IPython.utils.traits), 918
float_format (IPython.core.formatters.PlainTextFormatter attribute), 353
float_precision (IPython.core.formatters.PlainTextFormatter attribute), 353
flush() (IPython.core.displayhook.DisplayHook method), 320
flush() (IPython.lib.pretty.PrettyPrinter method), 583
flush() (IPython.lib.pretty.RepresentationPrinter method), 584
flush() (IPython.testing.globalipapp.StreamProxy method), 833
flush() (IPython.testing.mkdoctests.IndentOut method), 839
flush_finished() (IPython.lib.backgroundjobs.BackgroundJobManager method), 551
follow (IPython.parallel.client.view.LoadBalancedView attribute), 763
for_type() (in module IPython.lib.pretty), 584
for_type() (IPython.core.formatters.BaseFormatter method), 332
for_type() (IPython.core.formatters.HTMLFormatter method), 338
for_type() (IPython.core.formatters.JavascriptFormatter method), 344
for_type() (IPython.core.formatters.JSONFormatter method), 341
for_type() (IPython.core.formatters.LatexFormatter method), 347
for_type() (IPython.core.formatters.PlainTextFormatter method), 353
for_type() (IPython.core.formatters.PNGFormatter method), 349
for_type() (IPython.core.formatters.SVGFormatter method), 349
for_type_by_name() (in module IPython.lib.pretty), 584
for_type_by_name() (IPython.core.formatters.BaseFormatter method), 332
for_type_by_name() (IPython.core.formatters.HTMLFormatter method), 338
for_type_by_name() (IPython.core.formatters.JavascriptFormatter method), 344
for_type_by_name() (IPython.core.formatters.JSONFormatter method), 341
for_type_by_name() (IPython.core.formatters.LatexFormatter method), 347

for_type_by_name()
 (IPython.core.formatters.PlainTextFormatter
 method), 353

for_type_by_name()
 (IPython.core.formatters.PNGFormatter
 method), 350

for_type_by_name()
 (IPython.core.formatters.SVGFormatter
 method), 356

forget() (IPython.core.debugger.Pdb method), 310

format (IPython.utils.text.LSString attribute), 891

format() (IPython.core.formatters.DisplayFormatter
 method), 335

format() (IPython.utils.PyColorize.Parser method),
 849

format() (IPython.utils.text.EvalFormatter method),
 890

format2() (IPython.utils.PyColorize.Parser method),
 849

format_argspec() (in module IPython.core.oinspect),
 451

format_display_data() (in module IPython.core.formatters), 357

format_field() (IPython.utils.text.EvalFormatter
 method), 890

format_help() (IPython.config.loader.ArgumentParser
 method), 278

format_help() (IPython.core.magic_arguments.MagicArgumentParser
 method), 446

format_latex() (IPython.core.interactiveshell.InteractiveShell
 method), 381

format_latex() (IPython.core.magic.Magic method),
 421

format_screen() (in module IPython.utils.text), 897

format_stack_entry() (IPython.core.debugger.Pdb
 method), 310

format_type (IPython.core.formatters.BaseFormatter
 attribute), 333

format_type (IPython.core.formatters.FormatterABC
 attribute), 337

format_type (IPython.core.formatters.HTMLFormatter
 attribute), 339

format_type (IPython.core.formatters.JavascriptFormatter
 attribute), 344

format_type (IPython.core.formatters.JSONFormatter
 attribute), 341

format_type (IPython.core.formatters.LatexFormatter
 attribute), 347

format_type (IPython.core.formatters.PlainTextFormatter
 attribute), 353

format_type (IPython.core.formatters.PNGFormatter
 attribute), 350

format_type (IPython.core.formatters.SVGFormatter
 attribute), 356

format_types (IPython.core.formatters.DisplayFormatter
 attribute), 335

format_usage() (IPython.config.loader.ArgumentParser
 method), 278

format_usage() (IPython.core.magic_arguments.MagicArgumentParser
 method), 446

format_version() (IPython.config.loader.ArgumentParser
 method), 278

format_version() (IPython.core.magic_arguments.MagicArgumentParser
 method), 446

FormattedTB (class in IPython.core.ultrath), 540

formatter (IPython.parallel.apps.launcher.BatchSystemLauncher
 attribute), 640

formatter (IPython.parallel.apps.launcher.LSFControllerLauncher
 attribute), 646

formatter (IPython.parallel.apps.launcher.LSFEngineSetLauncher
 attribute), 650

formatter (IPython.parallel.apps.launcher.LSFLauncher
 attribute), 653

formatter (IPython.parallel.apps.launcher.PBSControllerLauncher
 attribute), 677

formatter (IPython.parallel.apps.launcher.PBSEngineSetLauncher
 attribute), 680

formatter (IPython.parallel.apps.launcher.PBSLauncher
 attribute), 684

formatter (IPython.parallel.apps.launcher.SGEControllerLauncher
 attribute), 688

formatter (IPython.parallel.apps.launcher.SGEEngineSetLauncher
 attribute), 691

formatter (IPython.parallel.apps.launcher.SGELauncher
 attribute), 694

FormatterABC (class in IPython.core.formatters),
 336

formatters (IPython.core.formatters.DisplayFormatter
 attribute), 335

forward_logging() (IPython.parallel.apps.ipcontrollerapp.IPCController
 method), 617

forward_logging() (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 623

forward_read_events() (in module IPython.parallel.apps.win32support),
 721

ForwarderThread (class in IPython.parallel.apps.win32support), [720](#)

freeze_term_title() (in module IPython.utils.terminal), [888](#)

fromkeys() (IPython.config.loader.Config static method), [279](#)

fromkeys() (IPython.parallel.client.client.Metadata static method), [751](#)

fromkeys() (IPython.parallel.util.Namespace static method), [826](#)

fromkeys() (IPython.parallel.util.ReverseDict static method), [827](#)

fromkeys() (IPython.testing.globalipapp.ipnsdict static method), [833](#)

fromkeys() (IPython.utils.coloransi.ColorSchemeTable generate_config_file() static method), [855](#)

fromkeys() (IPython.utils.ipstruct.Struct static method), [868](#)

full_path() (in module IPython.testing.tools), [846](#)

func (IPython.parallel.client.remotefunction.ParallelFunction attribute), [755](#)

func (IPython.parallel.client.remotefunction.RemoteFunction attribute), [755](#)

G

gather() (IPython.parallel.client.view.DirectView method), [758](#)

generate() (IPython.utils.traits.DefaultValueGenerator method), [915](#)

generate_config_file() (IPython.config.application.Application method), [264](#)

generate_config_file() (IPython.core.application.BaseIPythonApplication method), [289](#)

generate_config_file() (IPython.core.profileapp.ProfileApp method), [514](#)

generate_config_file() (IPython.core.profileapp.ProfileCreate method), [518](#)

generate_config_file() (IPython.core.profileapp.ProfileList method), [522](#)

generate_config_file() (IPython.parallel.apps.baseapp.BaseParallelApplication method), [589](#)

in generate_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterApp method), [595](#)

generate_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), [599](#)

generate_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), [605](#)

generate_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), [611](#)

generate_config_file() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp method), [617](#)

generate_exec_key() (in module IPython.parallel.util), [828](#)

generate_prompt() (in module IPython.core.hooks), [367](#)

get (IPython.config.loader.Config attribute), [279](#)

get (IPython.parallel.client.client.Metadata attribute), [751](#)

get (IPython.parallel.util.Namespace attribute), [826](#)

get (IPython.testing.globalipapp.ipnsdict attribute), [833](#)

get (IPython.utils.coloransi.ColorSchemeTable attribute), [855](#)

get (IPython.utils.ipstruct.Struct attribute), [868](#)

get() (AsyncResult method), [169](#)

get() (in module IPython.core.ipapi), [417](#)

get() (IPython.parallel.client.asyncresult.AsyncHubResult method), [740](#)

get() (IPython.parallel.client.asyncresult.AsyncMapResult method), [741](#)

get() (IPython.parallel.client.asyncresult.AsyncResult method), [742](#)

get() (IPython.parallel.client.view.DirectView method), [758](#)

get() (IPython.parallel.util.ReverseDict method), [827](#)

get() (IPython.utils.pickleshare.PickleShareDB method), [881](#)

| | | | |
|----------------------|--|----------------------|--|
| get_all_breaks() | (IPython.core.debugger.Pdb method), 310 | get_default_value() | (IPython.utils.traits.Enum method), 917 |
| get_app_qt4() | (in module IPython.lib.guisupport), 569 | get_default_value() | (IPython.utils.traits.Float method), 918 |
| get_app_wx() | (in module IPython.lib.guisupport), 569 | get_default_value() | (IPython.utils.traits.Instance method), 920 |
| get_break() | (IPython.core.debugger.Pdb method), 310 | get_default_value() | (IPython.utils.traits.Int method), 921 |
| get_breaks() | (IPython.core.debugger.Pdb method), 310 | get_default_value() | (IPython.utils.traits.List method), 923 |
| get_class_members() | (in module IPython.utils.dir2), 860 | get_default_value() | (IPython.utils.traits.Long method), 924 |
| get_default_colors() | (in module IPython.core.interactiveshell), 417 | get_default_value() | (IPython.utils.traits.ObjectName method), 925 |
| get_default_value() | (IPython.core.interactiveshell.Sep | get_default_value() | (IPython.utils.traits.Set method), 927 |
| method), 416 | ageDefaultValue() | | |
| get_default_value() | (IPython.utils.traits.Any method), 901 | get_default_value() | (IPython.utils.traits.TCPAddress method), 928 |
| get_default_value() | (IPython.utils.traits.Bool method), 902 | get_default_value() | (IPython.utils.traits.This method), 929 |
| get_default_value() | (IPython.utils.traits.Bytes method), 903 | get_default_value() | (IPython.utils.traits.TraitType method), 930 |
| get_default_value() | (IPython.utils.traits.CaselessStr method), 911 | get_default_value() | (IPython.utils.traits.Tuple method), 931 |
| get_default_value() | (IPython.utils.traits.CBool method), 904 | get_default_value() | (IPython.utils.traits.Type method), 933 |
| get_default_value() | (IPython.utils.traits.CBytes method), 905 | get_default_value() | (IPython.utils.traits.Unicode method), 934 |
| get_default_value() | (IPython.utils.traits.CComplex method), 906 | get_delims() | (IPython.core.completer.CompletionSplitter method), 299 |
| get_default_value() | (IPython.utils.traits.CFloat method), 907 | get_dict() | (IPython.parallel.client.asyncresult.AsyncHubResult method), 740 |
| get_default_value() | (IPython.utils.traits.CInt method), 908 | get_dict() | (IPython.parallel.client.asyncresult.AsyncMapResult method), 741 |
| get_default_value() | (IPython.utils.traits.ClassBasedTraitType method), 912 | get_dict() | (IPython.parallel.client.asyncresult.AsyncResult method), 742 |
| get_default_value() | (IPython.utils.traits.CLong method), 909 | get_env_vars() | (IPython.parallel.apps.winhpcjob.IPControllerTask method), 726 |
| get_default_value() | (IPython.utils.traits.Complex method), 913 | get_env_vars() | (IPython.parallel.apps.winhpcjob.IPEngineTask method), 731 |
| get_default_value() | (IPython.utils.traits.Container method), 914 | get_env_vars() | (IPython.parallel.apps.winhpcjob.WinHPCTask method), 737 |
| get_default_value() | (IPython.utils.traits.CUnicode method), 910 | get_exception_only() | (IPython.core.ultratb.AutoFormattedTB method), 538 |
| get_default_value() | (IPython.utils.traits.Dict method), 915 | get_exception_only() | (IPython.core.ultratb.ColorTB method), 539 |
| get_default_value() | (IPython.utils.traits.DottedObjectName method), 916 | get_exception_only() | |

(IPython.core.ultratb.FormattedTB
method), 541

get_exception_only() (IPython.core.ultratb.ListTB
method), 542

get_exception_only()
(IPython.core.ultratb.SyntaxTB method),
544

get_extra_args() (IPython.config.loader.ArgParseConfig
method), 277

get_field() (IPython.utils.text.EvalFormatter
method), 890

get_file_breaks() (IPython.core.debugger.Pdb
method), 310

get_handler_by_esc()
(IPython.core.prefilter.PrefilterManager
method), 500

get_handler_by_name()
(IPython.core.prefilter.PrefilterManager
method), 500

get_history() (IPython.parallel.controller.dictdb.DictDB
method), 779

get_history() (IPython.parallel.controller.hub.Hub
method), 788

get_history() (IPython.parallel.controller.sqlitedb.SQLiteDB
method), 801

get_home_dir() (in module IPython.utils.path), 878

get_input_encoding() (in module
IPython.core.inputsplitter), 373

get_ipython() (in module
IPython.testing.globalipapp), 834

get_ipython() (IPython.core.interactiveshell.InteractiveShell
method), 382

get_ipython_dir() (in module IPython.utils.path),
879

get_ipython_module_path() (in module
IPython.utils.path), 879

get_ipython_package_dir() (in module
IPython.utils.path), 879

get_list() (IPython.utils.text.LSString method), 891

get_list() (IPython.utils.text.SList method), 896

get_long_path_name() (in module
IPython.utils.path), 879

get_metadata() (IPython.core.interactiveshell.SeparateGetContext
method), 416

get_metadata() (IPython.utils.traits.Any method),
901

get_metadata() (IPython.utils.traits.Bool method),
902

get_metadata() (IPython.utils.traits.Bytes
method), 903

get_metadata() (IPython.utils.traits.CaselessStrEnum
method), 911

get_metadata() (IPython.utils.traits.CBool
method), 904

get_metadata() (IPython.utils.traits.CBytes
method), 905

get_metadata() (IPython.utils.traits.CComplex
method), 906

get_metadata() (IPython.utils.traits.CFloat
method), 907

get_metadata() (IPython.utils.traits.CInt method),
908

get_metadata() (IPython.utils.traits.ClassBasedTraitType
method), 912

get_metadata() (IPython.utils.traits.CLong
method), 909

get_metadata() (IPython.utils.traits.Complex
method), 913

get_metadata() (IPython.utils.traits.Container
method), 914

get_metadata() (IPython.utils.traits.CUnicode
method), 910

get_metadata() (IPython.utils.traits.Dict method),
915

get_metadata() (IPython.utils.traits.DottedObjectName
method), 916

get_metadata() (IPython.utils.traits.Enum
method), 917

get_metadata() (IPython.utils.traits.Float method),
918

get_metadata() (IPython.utils.traits.Instance
method), 921

get_metadata() (IPython.utils.traits.Int method),
921

get_metadata() (IPython.utils.traits.List method),
923

get_metadata() (IPython.utils.traits.Long
method), 924

get_metadata() (IPython.utils.traits.ObjectName
method), 925

get_metadata() (IPython.utils.traits.Set method),
927

get_metadata() (IPython.utils.traits.TCPAddress
method), 928

get_metadata() (IPython.utils.traits.This method),
929

get_metadata() (IPython.utils.traits.TraitType method), 930
get_metadata() (IPython.utils.traits.Tuple method), 932
get_metadata() (IPython.utils.traits.Type method), 933
get_metadata() (IPython.utils.traits.Unicode method), 934
get_names() (IPython.core.debugger.Pdb method), 310
get_nlstr() (IPython.utils.text.LSString method), 891
get_nlstr() (IPython.utils.text.SList method), 896
get_pager_cmd() (in module IPython.core.page), 452
get_pager_start() (in module IPython.core.page), 452
get_paths() (IPython.utils.text.LSString method), 891
get_paths() (IPython.utils.text.SList method), 896
get_pid_from_file() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 589
get_pid_from_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 599
get_pid_from_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 605
get_pid_from_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 611
get_pid_from_file() (IPython.parallel.apps.ipcontrollerapp.IPClusterEngines method), 617
get_pid_from_file() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 623
get_pid_from_file() (IPython.parallel.apps.iploggerapp.IPLoggerApp method), 632
get_plugin() (IPython.core.plugin.PluginManager method), 459
get_py_filename() (in module IPython.utils.path), 879
get_pyos_inpthook() (IPython.lib.inpthook.InputHookManager method), 572
get_pyos_inpthook_as_func() (IPython.lib.inpthook.InputHookManager method), 572
get_range() (IPython.core.history.HistoryManager method), 360
get_range_by_str() (IPython.core.history.HistoryManager method), 360
get_readline_tail() (IPython.core.interactiveshell.Readline method), 415
get_record() (IPython.parallel.controller.dictdb.DictDB method), 780
get_record() (IPython.parallel.controller.sqlitedb.SQLiteDB method), 801
get_result() (IPython.parallel.client.client.Client method), 747
get_result() (IPython.parallel.client.view.DirectView method), 758
get_result() (IPython.parallel.client.view.LoadBalancedView method), 763
get_result() (IPython.parallel.client.view.View method), 769
get_results() (IPython.parallel.controller.hub.Hub method), 788
get_root_modules() (in module IPython.core.completerlib), 303
get_slice() (in module IPython.utils.data), 859
get_spstr() (IPython.utils.text.LSString method), 896
get_spstr() (IPython.utils.text.SList method), 896
get_star() (IPython.core.debugger.Pdb method), 310
get_star() (IPython.core.history.HistoryManager method), 361
get_terminal_size() (in module IPython.utils.terminal), 888
getargspec() (in module IPython.core.oinspect), 451
get_ipcontrollerapp() (IPython.utils.text.EvalFormatter method), 890
get_ipengineapp() (IPython.utils.path method), 879
get_iploggerapp() (IPython.utils.newserialized.ISerialized method), 873
get_pluginmanager() (IPython.core.plugin.PluginManager method), 874
get_data() (IPython.utils.newserialized.Serialized method), 874
get_data() (IPython.utils.newserialized.SerializeIt method), 874
get_data_size() (IPython.utils.newserialized.ISerialized method), 873
get_data_size() (IPython.utils.newserialized.Serialized method), 874
get_data_size() (IPython.utils.newserialized.SerializeIt method), 874
get_doc() (in module IPython.core.oinspect), 451
getfigs() (in module IPython.lib.pylabtools), 585
gethashfile() (in module IPython.utils.pickleshare), 882
getpickle() (IPython.utils.pickleshare.PickleShareDB method), 415

method), 881
getmembers() (in module IPython.utils.traitlets), 934
getMetadata() (IPython.utils.newserialized.ISerialized
 method), 873
getMetadata() (IPython.utils.newserialized.Serialized
 method), 874
getMetadata() (IPython.utils.newserialized.SerializeIt
 method), 874
getName() (IPython.core.history.HistorySavingThread
 method), 363
getName() (IPython.lib.backgroundjobs.BackgroundJob
 method), 548
getName() (IPython.lib.backgroundjobs.BackgroundJob
 method), 549
getName() (IPython.lib.backgroundjobs.BackgroundJobFunc
 method), 550
getName() (IPython.parallel.apps.win32support.ForwarderThread
 method), 720
getObject() (IPython.utils.newserialized.IUnSerialized
 method), 873
getObject() (IPython.utils.newserialized.UnSerialized
 method), 875
getObject() (IPython.utils.newserialized.UnSerializeIt
 method), 874
getObject() (IPython.utils.pickleutil.CannedFunction
 method), 883
getObject() (IPython.utils.pickleutil.CannedObject
 method), 883
getObject() (IPython.utils.pickleutil.Reference
 method), 883
getoutput() (IPython.core.interactiveshell.InteractiveShell
 method), 382
getPartition() (IPython.parallel.client.map.Map
 method), 753
getPartition() (IPython.parallel.client.map.RoundRobinMap
 method), 753
getsource() (in module IPython.core.oinspect), 452
getTestCaseNames() (in module
 IPython.testing.nosepatch), 840
getTypeDescriptor()
 (IPython.utils.newserialized.ISerialized
 method), 873
getTypeDescriptor()
 (IPython.utils.newserialized.Serialized
 method), 874
getTypeDescriptor()
 (IPython.utils.newserialized.SerializeIt
 method), 874
global_matches() (IPython.core.completer.Completer
 method), 299
global_matches() (IPython.core.completer.IPCCompleter
 method), 301
graph (IPython.parallel.controller.scheduler.TaskScheduler
 attribute), 797
Green (IPython.utils.coloransi.InputTermColors at-
 tribute), 857
Green (IPython.utils.coloransi.TermColors at-
 tribute), 858
grep() (IPython.utils.text.SList method), 896
GroupApp (class in IPython.lib.pretty), 582
group() (IPython.lib.pretty.PrettyPrinter method),
 583
group() (IPython.lib.pretty.RepresentationPrinter
 method), 584
GroupQueue (class in IPython.lib.pretty), 582
H
handle() (IPython.core.prefilter.AliasHandler
 method), 464
handle() (IPython.core.prefilter.AutoHandler
 method), 472
handle() (IPython.core.prefilter.EmacsHandler
 method), 479
handle() (IPython.core.prefilter.HelpHandler
 method), 482
handle() (IPython.core.prefilter.MacroHandler
 method), 490
handle() (IPython.core.prefilter.MagicHandler
 method), 492
handle() (IPython.core.prefilter.PrefilterHandler
 method), 498
handleMap() (IPython.core.prefilter.ShellEscapeHandler
 method), 511
handle_command_def() (IPython.core.debugger.Pdb
 method), 310
handle_heart_failure()
 (IPython.parallel.controller.heartmonitor.HeartMonitor
 method), 782
handle_heart_failure()
 (IPython.parallel.controller.hub.Hub
 method), 788
handle_new_heart() (IPython.parallel.controller.heartmonitor.HeartM
 method), 782
handle_new_heart() (IPython.parallel.controller.hub.Hub
 method), 788

handle_pong() (IPython.parallel.controller.heartmonitor.HeartMonitor method), 782
 handle_result() (IPython.parallel.controller.scheduler.TaskScheduler method), 797
 handle_stderr() (IPython.parallel.apps.launcher.IPClusterLauncher method), 643
 handle_stderr() (IPython.parallel.apps.launcher.LocalClusterLauncher method), 657
 handle_stderr() (IPython.parallel.apps.launcher.LocalEnginesLauncher method), 546
 handle_stderr() (IPython.parallel.apps.launcher.LocalPackerLauncher method), 665
 handle_stderr() (IPython.parallel.apps.launcher.MPIExecutorLauncher method), 668
 handle_stderr() (IPython.parallel.apps.launcher.MPIExecutiveLauncher method), 671
 handle_stderr() (IPython.parallel.apps.launcher.MPIExecutiveHandler method), 674
 handle_stderr() (IPython.parallel.apps.launcher.SSHControllerLauncher method), 697
 handle_stderr() (IPython.parallel.apps.launcher.SSHEngineLauncher method), 700
 handle_stderr() (IPython.parallel.apps.launcher.SSHHandler method), 706
 handle_stdout() (IPython.parallel.apps.launcher.IPClusterHandler method), 643
 handle_stdout() (IPython.parallel.apps.launcher.LocalController method), 657
 handle_stdout() (IPython.parallel.apps.launcher.LocalEnginesHandler method), 660
 handle_stdout() (IPython.parallel.apps.launcher.LocalPackerHandler method), 665
 handle_stdout() (IPython.parallel.apps.launcher.MPIExecutiveHandler method), 668
 handle_stdout() (IPython.parallel.apps.launcher.MPIExecutiveKeyHandler method), 671
 handle_stdout() (IPython.parallel.apps.launcher.MPIExecutiveKeyHandler method), 674
 handle_stdout() (IPython.parallel.apps.launcher.SSHControllerHandler method), 698
 handle_stdout() (IPython.parallel.apps.launcher.SSHEngineHandler method), 700
 handle_stdout() (IPython.parallel.apps.launcher.SSHHandler method), 706
 handle_stranded_tasks()
 handle_unmet_dependency()

(IPython.parallel.controller.scheduler.TaskScheduler method), 797
 handle_key() (IPython.utils.pickleshare.PickleShareDB method), 881
 has_open_quotes() (in module)

IPython.core.completer), 302
hasattr() (IPython.utils.ipstruct.Struct method), 869
HasTraits (class in IPython.utils.traits), 919
hb (IPython.parallel.controller.hub.HubFactory attribute), 792
hcompress() (IPython.utils.pickleshare.PickleShareDB method), 881
hdict() (IPython.utils.pickleshare.PickleShareDB method), 881
Heart (class in IPython.parallel.controller.heartmonitor), 781
heartbeat (IPython.parallel.controller.hub.EngineConnector attribute), 785
HeartMonitor (class in IPython.parallel.controller.heartmonitor), 781
heartmonitor (IPython.parallel.controller.hub.Hub attribute), 788
heartmonitor (IPython.parallel.controller.hub.HubFactoattribute), 792
hearts (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 782
hearts (IPython.parallel.controller.hub.Hub attribute), 788
help_a() (IPython.core.debugger.Pdb method), 310
help_alias() (IPython.core.debugger.Pdb method), 310
help_args() (IPython.core.debugger.Pdb method), 310
help_b() (IPython.core.debugger.Pdb method), 310
help_break() (IPython.core.debugger.Pdb method), 310
help_bt() (IPython.core.debugger.Pdb method), 310
help_c() (IPython.core.debugger.Pdb method), 310
help_ci() (IPython.core.debugger.Pdb method), 310
help_clear() (IPython.core.debugger.Pdb method), 310
help_commands() (IPython.core.debugger.Pdb method), 310
help_condition() (IPython.core.debugger.Pdb method), 310
help_cont() (IPython.core.debugger.Pdb method), 310
help_continue() (IPython.core.debugger.Pdb method), 310
help_d() (IPython.core.debugger.Pdb method), 310
help_debug() (IPython.core.debugger.Pdb method), 310
help_disable() (IPython.core.debugger.Pdb method), 310
help_down() (IPython.core.debugger.Pdb method), 310
help_enable() (IPython.core.debugger.Pdb method), 310
help_EOF() (IPython.core.debugger.Pdb method), 310
help_exec() (IPython.core.debugger.Pdb method), 311
help_exit() (IPython.core.debugger.Pdb method), 311
help_h() (IPython.core.debugger.Pdb method), 311
help_help() (IPython.core.debugger.Pdb method), 311
help_ignore() (IPython.core.debugger.Pdb method), 311
help_j() (IPython.core.debugger.Pdb method), 311
help_jump() (IPython.core.debugger.Pdb method), 311
help_list() (IPython.core.debugger.Pdb method), 311
help_n() (IPython.core.debugger.Pdb method), 311
help_next() (IPython.core.debugger.Pdb method), 311
help_p() (IPython.core.debugger.Pdb method), 311
help_pdb() (IPython.core.debugger.Pdb method), 311
help_pp() (IPython.core.debugger.Pdb method), 311
help_q() (IPython.core.debugger.Pdb method), 311
help_quit() (IPython.core.debugger.Pdb method), 311
help_r() (IPython.core.debugger.Pdb method), 311
help_restart() (IPython.core.debugger.Pdb method), 311
help_return() (IPython.core.debugger.Pdb method), 311
help_run() (IPython.core.debugger.Pdb method), 311
help_s() (IPython.core.debugger.Pdb method), 311
help_step() (IPython.core.debugger.Pdb method), 311
help_tbreak() (IPython.core.debugger.Pdb method), 311
help_u() (IPython.core.debugger.Pdb method), 311
help_unalias() (IPython.core.debugger.Pdb method), 311
help_unt() (IPython.core.debugger.Pdb method),

311
help_until() (IPython.core.debugger.Pdb method), 311
help_up() (IPython.core.debugger.Pdb method), 311
help_w() (IPython.core.debugger.Pdb method), 311
help_whatis() (IPython.core.debugger.Pdb method), 311
help_where() (IPython.core.debugger.Pdb method), 311
HelpHandler (class in IPython.core.prefilter), 482
hget() (IPython.utils.pickleshare.PickleShareDB method), 881
hist_file (IPython.core.history.HistoryManager attribute), 361
history (IPython.parallel.client.client.Client attribute), 748
history (IPython.parallel.client.view.DirectView attribute), 758
history (IPython.parallel.client.view.LoadBalancedView attribute), 764
history (IPython.parallel.client.view.View attribute), 769
history_length (IPython.core.interactiveshell.InteractiveShell attribute), 382
history_manager (IPython.core.interactiveshell.InteractiveShell attribute), 382
HistoryManager (class in IPython.core.history), 359
HistorySavingThread (class in IPython.core.history), 363
HomeDirError (class in IPython.utils.path), 877
hook (IPython.core.display_trap.DisplayTrap attribute), 317
hostname (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 698
hostname (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 701
hostname (IPython.parallel.apps.launcher.SSHLauncher attribute), 706
hset() (IPython.utils.pickleshare.PickleShareDB method), 881
HTMLFormatter (class in IPython.core.formatters), 337
Hub (class in IPython.parallel.controller.hub), 786
hub_history() (IPython.parallel.client.client.Client method), 748
HubFactory (class in IPython.parallel.controller.hub), 791
hwm (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
|
id (IPython.parallel.controller.heartmonitor.Heart attribute), 781
id (IPython.parallel.controller.hub.EngineConnector attribute), 785
id (IPython.parallel.engine.engine.EngineFactory attribute), 804
ident (IPython.core.history.HistorySavingThread attribute), 363
ident (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 548
ident (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 549
ident (IPython.lib.backgroundjobs.BackgroundJobFunc attribute), 550
ident (IPython.parallel.apps.win32support.ForwarderThread attribute), 720
ident (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
ident (IPython.parallel.engine.engine.EngineFactory InteractiveShell attribute), 804
ident (IPython.parallel.engine.streamkernel.Kernel InteractiveShell attribute), 810
identchars (IPython.core.debugger.Pdb attribute), 311
idgrep() (in module IPython.utils.text), 897
IdInUse (class in IPython.parallel.error), 816
ids (IPython.parallel.client.client.Client attribute), 748
ids (IPython.parallel.controller.hub.Hub attribute), 788
igrep() (in module IPython.utils.text), 897
imap (IPython.parallel.client.view.DirectView method), 758
imap() (IPython.parallel.client.view.LoadBalancedView method), 764
imap() (IPython.parallel.client.view.View method), 769
import_fail_info() (in module IPython.utils.attic), 850
import_item() (in module IPython.utils.importstring), 864
import_module() (in module IPython.lib.deepcopy), 554
import_pylab() (in module IPython.lib.pylabtools), 585

import_statements (IPython.parallel.apps.ipcontrollerapi.PCIPythonApp traitlets.Float method), 918
 attribute), 617
importer (IPython.parallel.client.view.DirectView attribute), 758
ImpossibleDependency (class in IPython.parallel.error), 816
incoming_registrations (IPython.parallel.controller.hub.Hub attribute), 788
indent() (in module IPython.parallel.apps.winhpcjob), 739
indent() (in module IPython.utils.text), 897
indent() (IPython.lib.pretty.PrettyPrinter method), 583
indent() (IPython.lib.pretty.RepresentationPrinter method), 584
indent_spaces (IPython.core.inputsplitter.InputSplitter attribute), 371
indent_spaces (IPython.core.inputsplitter.IPythonInputSplitter attribute), 370
IndentOut (class in IPython.testing.mkdoctests), 839
index (IPython.utils.text.LSString attribute), 891
index (IPython.utils.text.SList attribute), 896
info() (in module IPython.utils.warn), 936
info() (IPython.core.interactiveshell.SeparateUnicode method), 416
info() (IPython.core.oinspect.Inspector method), 449
info() (IPython.utils.traitlets.Any method), 901
info() (IPython.utils.traitlets.Bool method), 902
info() (IPython.utils.traitlets.Bytes method), 903
info() (IPython.utils.traitlets.CaselessStrEnum method), 911
info() (IPython.utils.traitlets.CBool method), 904
info() (IPython.utils.traitlets.CBytes method), 905
info() (IPython.utils.traitlets.CComplex method), 906
info() (IPython.utils.traitlets.CFloat method), 907
info() (IPython.utils.traitlets.CInt method), 908
info() (IPython.utils.traitlets.ClassBasedTraitType method), 912
info() (IPython.utils.traitlets.CLong method), 909
info() (IPython.utils.traitlets.Complex method), 913
info() (IPython.utils.traitlets.Container method), 914
info() (IPython.utils.traitlets.CUnicode method), 910
info() (IPython.utils.traitlets.Dict method), 915
info() (IPython.utils.traitlets.DottedObjectName method), 916
info() (IPython.utils.traitlets.Enum method), 917
info() (IPython.utils.traitlets.Float method), 918
 attribute), 918
info() (IPython.utils.traitlets.Instance method), 921
info() (IPython.utils.traitlets.Int method), 921
info() (IPython.utils.traitlets.List method), 923
info() (IPython.utils.traitlets.Long method), 924
info() (IPython.utils.traitlets.ObjectName method), 925
info() (IPython.utils.traitlets.Set method), 927
info() (IPython.utils.traitlets.TCPAddress method), 928
info() (IPython.utils.traitlets.This method), 929
info() (IPython.utils.traitlets.TraitType method), 930
info() (IPython.utils.traitlets.Tuple method), 932
info() (IPython.utils.traitlets.Type method), 933
info() (IPython.utils.traitlets.Unicode method), 934
info_text (IPython.core.interactiveshell.SeparateUnicode attribute), 416
info_text (IPython.utils.traitlets.Any attribute), 901
Splitter (IPython.utils.traitlets.Bool attribute), 902
info_text (IPython.utils.traitlets.Bytes attribute), 903
info_text (IPython.utils.traitlets.CaselessStrEnum attribute), 911
info_text (IPython.utils.traitlets.CBool attribute), 904
info_text (IPython.utils.traitlets.CBytes attribute), 905
info_text (IPython.utils.traitlets.CComplex attribute), 906
info_text (IPython.utils.traitlets.CFloat attribute), 907
info_text (IPython.utils.traitlets.CInt attribute), 908
info_text (IPython.utils.traitlets.ClassBasedTraitType attribute), 912
info_text (IPython.utils.traitlets.CLong attribute), 909
info_text (IPython.utils.traitlets.Complex attribute), 913
info_text (IPython.utils.traitlets.Container attribute), 914
info_text (IPython.utils.traitlets.CUnicode attribute), 910
info_text (IPython.utils.traitlets.Dict attribute), 916
info_text (IPython.utils.traitlets.DottedObjectName attribute), 916
info_text (IPython.utils.traitlets.Enum attribute), 917
info_text (IPython.utils.traitlets.Float attribute), 918
info_text (IPython.utils.traitlets.Instance attribute), 921

info_text (IPython.utils.traits.Int attribute), 921
 info_text (IPython.utils.traits.List attribute), 923
 info_text (IPython.utils.traits.Long attribute), 924
 info_text (IPython.utils.traits.ObjectName attribute), 925
 info_text (IPython.utils.traits.Set attribute), 927
 info_text (IPython.utils.traits.TCPAddress attribute), 928
 info_text (IPython.utils.traits.This attribute), 929
 info_text (IPython.utils.traits.TraitType attribute), 930
 info_text (IPython.utils.traits.Tuple attribute), 932
 info_text (IPython.utils.traits.Type attribute), 933
 info_text (IPython.utils.traits.Unicode attribute), 934
 init() (IPython.core.interactiveshell.SeparateUnicode method), 416
 init() (IPython.utils.traits.Any method), 901
 init() (IPython.utils.traits.Bool method), 902
 init() (IPython.utils.traits.Bytes method), 903
 init() (IPython.utils.traits.CaselessStrEnum method), 911
 init() (IPython.utils.traits.CBool method), 904
 init() (IPython.utils.traits.CBytes method), 905
 init() (IPython.utils.traits.CComplex method), 906
 init() (IPython.utils.traits.CFloat method), 907
 init() (IPython.utils.traits.CInt method), 908
 init() (IPython.utils.traits.ClassBasedTraitType method), 912
 init() (IPython.utils.traits.CLong method), 909
 init() (IPython.utils.traits.Complex method), 913
 init() (IPython.utils.traits.Container method), 914
 init() (IPython.utils.traits.CUnicode method), 910
 init() (IPython.utils.traits.Dict method), 916
 init() (IPython.utils.traits.DottedObjectName method), 916
 init() (IPython.utils.traits.Enum method), 917
 init() (IPython.utils.traits.Float method), 918
 init() (IPython.utils.traits.Instance method), 921
 init() (IPython.utils.traits.Int method), 921
 init() (IPython.utils.traits.List method), 923
 init() (IPython.utils.traits.Long method), 924
 init() (IPython.utils.traits.ObjectName method), 925
 init() (IPython.utils.traits.Set method), 927
 init() (IPython.utils.traits.TCPAddress method), 928
 init() (IPython.utils.traits.This method), 929
 init() (IPython.utils.traits.TraitType method), 930
 init() (IPython.utils.traits.Tuple method), 932
 init() (IPython.utils.traits.Type method), 933
 init() (IPython.utils.traits.Unicode method), 934
 init_alias() (IPython.core.interactiveshell.InteractiveShell method), 382
 init_aliases() (IPython.core.alias.AliasManager method), 285
 init_builtins() (IPython.core.interactiveshell.InteractiveShell method), 382
 init_checkers() (IPython.core.prefilter.PrefilterManager method), 501
 init_code() (IPython.core.shellapp.InteractiveShellApp method), 534
 init_completer() (IPython.core.interactiveshell.InteractiveShell method), 382
 init_config_files() (IPython.core.application.BaseIPythonApplication method), 289
 init_config_files() (IPython.core.profileapp.ProfileCreate method), 518
 init_config_files() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 589
 init_config_files() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 599
 init_config_files() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 605
 init_config_files() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 611
 init_config_files() (IPython.parallel.apps.ipcontrollerapp.IPController method), 617
 init_config_files() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 623
 init_config_files() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 632
 init_crash_handler() (IPython.core.application.BaseIPythonApplication method), 289
 init_crash_handler() (IPython.core.profileapp.ProfileCreate method), 518
 init_crash_handler() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 589
 init_crash_handler() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 599
 init_crash_handler() (IPython.parallel.apps.ipclusterapp.IPClusterStart

method), 605
init_crash_handler()
 (IPython.parallel.apps.ipclusterapp.IPClusterSmp_io() (IPython.core.interactiveshell.InteractiveShell
 method), 611
init_crash_handler()
 (IPython.parallel.apps.ipcontrollerapp.IPControlleipython_dir() (IPython.core.interactiveshell.InteractiveShell
 method), 617
init_crash_handler()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 623
init_crash_handler()
 (IPython.parallel.apps.iploggerapp.IPLoaderApp_logger() (IPython.core.interactiveshell.InteractiveShell
 method), 632
init_create_namespaces()
 (IPython.core.interactiveshell.InteractiveShell
 method), 382
init_db()
 (IPython.core.history.HistoryManager
 method), 361
init_display_formatter()
 (IPython.core.interactiveshell.InteractiveShellinit_logging() (IPython.core.profileapp.ProfileApp
 method), 382
init_display_pub()
 (IPython.core.interactiveshell.InteractiveShellinit_Sigiling() (IPython.core.profileapp.ProfileList
 method), 382
init_displayhook()
 (IPython.core.interactiveshell.InteractiveShellinit_Sigiling() (IPython.parallel.apps.baseapp.BaseParallelApplication
 method), 382
init_encoding()
 (IPython.core.interactiveshell.InteractiveShelllogging() (IPython.parallel.apps.ipclusterapp.IPClusterApp
 method), 382
init_engine()
 (IPython.parallel.apps.ipengineapp.IPEngineinit_Appging() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 method), 624
init_environment()
 (IPython.core.interactiveshell.InteractiveShellinit_Shellhg() (IPython.parallel.apps.ipclusterapp.IPClusterStart
 method), 382
init_extension_manager()
 (IPython.core.interactiveshell.InteractiveShell
 method), 382
init_extensions()
 (IPython.core.shellapp.InteractiveShellApp
 method), 534
init_handlers()
 (IPython.core.prefilter.PrefilterManager
 method), 501
init_history()
 (IPython.core.interactiveshell.InteractiveShell
 method), 382
init_hooks()
 (IPython.core.interactiveshell.InteractiveShell
 method), 382
init_hub()
 (IPython.parallel.apps.ipcontrollerapp.IPControllerAppmethod), 383
init_hub()
 (IPython.parallel.controller.hub.HubFactory
 method), 792
init_inspector()
 (IPython.core.interactiveshell.InteractiveShell
 method), 383
init_instance_attrs()
 (IPython.core.interactiveshell.InteractiveShell
 method), 383
init_ipython()
 (IPython.core.history), 364
init_launchers()
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 method), 599
init_launchers()
 (IPython.parallel.apps.ipclusterapp.IPClusterStart
 method), 605
init_logging()
 (IPython.config.application.Application
 method), 264
init_logging()
 (IPython.core.application.BaseIPythonApplication
 method), 289
init_logging()
 (IPython.core.profileapp.ProfileApp
 method), 514
init_logging()
 (IPython.core.profileapp.ProfileCreate
 method), 518
init_logging()
 (IPython.core.profileapp.ProfileList
 method), 522
init_logging()
 (IPython.parallel.apps.baseapp.BaseParallelApplication
 method), 589
init_logging()
 (IPython.parallel.apps.ipclusterapp.IPClusterApp
 method), 595
init_logging()
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 method), 599
init_logging()
 (IPython.parallel.apps.ipclusterapp.IPClusterStart
 method), 605
init_logging()
 (IPython.parallel.apps.ipclusterapp.IPClusterStop
 method), 611
init_logging()
 (IPython.parallel.apps.ipcontrollerapp.IPControllerApp
 method), 617
init_logging()
 (IPython.parallel.apps.iploggerapp.IPLoaderApp
 method), 617
init_logstart()
 (IPython.core.interactiveshell.InteractiveShell
 method), 632
init_magics()
 (IPython.core.interactiveshell.InteractiveShell
 method), 383
init_mpi()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 624
init_payload()
 (IPython.core.interactiveshell.InteractiveShell
 method), 624
init_pdb()
 (IPython.core.interactiveshell.InteractiveShell
 method), 383

```

        method), 383
init_plugin_manager()
    (IPython.core.interactiveshell.InteractiveShell
        method), 383
init_prefilter() (IPython.core.interactiveshell.InteractiveShell
    method), 383
init_profile_dir() (IPython.core.application.BaseIPythonApplication
    method), 289
init_profile_dir() (IPython.core.interactiveshell.InteractiveShell
    method), 383
init_profile_dir() (IPython.core.profileapp.ProfileCreate
    method), 518
init_profile_dir() (IPython.parallel.apps.baseapp.BaseParallelApplication
    method), 589
init_profile_dir() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
    method), 599
init_profile_dir() (IPython.parallel.apps.ipclusterapp.IPClusterStart
    method), 605
init_profile_dir() (IPython.parallel.apps.ipclusterapp.IPClusterStop
    method), 611
init_profile_dir() (IPython.parallel.apps.ipcontrollerapp.IPClusterController
    method), 617
init_profile_dir() (IPython.parallel.apps.ipengineapp.IPEngineApp
    method), 624
init_profile_dir() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
    method), 632
init_prompts() (IPython.core.interactiveshell.InteractiveShell
    method), 383
init_pushd_popd_magic()
    (IPython.core.interactiveshell.InteractiveShell
        method), 383
init_readline() (IPython.core.interactiveshell.InteractiveShell
    method), 383
init_record()
    (in module
        IPython.parallel.controller.hub), 794
init_reload_doctest()
    (IPython.core.interactiveshell.InteractiveShell
        method), 383
init_schedulers() (IPython.parallel.apps.ipcontrollerapp.IPClusterController
    method), 617
init_script (IPython.parallel.apps.ipengineapp.MPI
    attribute), 628
init_shell() (IPython.core.shellapp.InteractiveShellApp
    method), 534
init_signal() (IPython.parallel.apps.ipclusterapp.IPClusterEngine
    method), 599
init_signal() (IPython.parallel.apps.ipclusterapp.IPClusterStart
    method), 605
init_syntax_highlighting()
    (IPython.core.interactiveshell.InteractiveShell
        method), 383
init_sys_modules() (IPython.core.interactiveshell.InteractiveShell
    method), 383
init_traceback_handlers()
initShells() (IPython.core.prefilter.PrefilterManager
    method), 501
init_user_ns() (IPython.core.interactiveshell.InteractiveShell
    method), 383
initUserNS() (IPython.core.config.application.Application
    method), 265
initUserNS() (IPython.core.config.application.BaseIPythonApplication
    method), 289
initUserNS() (IPython.core.profileapp.ProfileApp
    method), 514
initUserNS() (IPython.parallel.apps.baseapp.BaseParallelApplication
    method), 589
initialize() (IPython.parallel.apps.ipclusterapp.IPClusterApp
    method), 595
initialize() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
    method), 600
initialize() (IPython.parallel.apps.ipclusterapp.IPClusterStart
    method), 605
initialize() (IPython.parallel.apps.ipclusterapp.IPClusterStop
    method), 611
initialize() (IPython.parallel.apps.ipcontrollerapp.IPClusterController
    method), 617
initialize() (IPython.parallel.apps.ipengineapp.IPEngineApp
    method), 624
initialize() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
    method), 632
initialize_subcommand()
    (IPython.config.application.Application
        method), 265
initialize_subcommand()
    (IPython.core.interactiveshell.InteractiveShell
        method), 624
initialize_subcommand()
    (IPython.core.config.application.BaseIPythonApplication
        method), 289
initialize_subcommand()
    (IPython.core.profileapp.ProfileApp
        method), 289

```

method), 514
initialize_subcommand()
 (IPython.core.profileapp.ProfileCreate
 method), 518
initialize_subcommand()
 (IPython.core.profileapp.ProfileList
 method), 522
initialize_subcommand()
 (IPython.parallel.apps.baseapp.BaseParallelApplication
 method), 589
initialize_subcommand()
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 method), 595
initialize_subcommand()
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 method), 600
initialize_subcommand()
 (IPython.parallel.apps.ipclusterapp.IPClusterStop
 method), 605
initialize_subcommand()
 (IPython.parallel.apps.ipclusterapp.IPClusterStop
 method), 611
initialize_subcommand()
 (IPython.parallel.apps.ipcontrollerapp.IPClusterStop
 method), 617
initialize_subcommand()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 624
initialize_subcommand()
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 method), 632
input_hist_parsed (IPython.core.history.HistoryManager
 attribute), 361
input_hist_raw (IPython.core.history.HistoryManager
 attribute), 361
input_mode (IPython.core.inputsplitter.InputSplitter
 attribute), 371
input_mode (IPython.core.inputsplitter.IPythonInputSplitter
 attribute), 370
input_prefilter() (in module IPython.core.hooks),
 367
initialize_subcommand()
 (IPython.parallel.apps.ipcontrollerapp.IPClusterStop
 method), 617
initialize_subcommand()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 624
initialize_subcommand()
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 method), 632
initialized() (IPython.config.application.Application
 class method), 265
initialized() (IPython.config.configurable.SingletonConfigurableObject)
 (class in IPython.utils.generics),
 274
initialized() (IPython.core.application.BaseIPythonApplication
 class method), 289
initialized() (IPython.core.interactiveshell.InteractiveShell
 class method), 383
initialized() (IPython.core.profileapp.ProfileApp
 class method), 514
initialized() (IPython.core.profileapp.ProfileCreate
 class method), 518
initialized() (IPython.core.profileapp.ProfileList
 class method), 522
initialized() (IPython.parallel.apps.baseapp.BaseParallelApplication
 class method), 589
initialized() (IPython.parallel.apps.ipclusterapp.IPClusterStop
 class method), 595
class method), 595
initialized() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
 class method), 600
initialized() (IPython.parallel.apps.ipclusterapp.IPClusterStart
 class method), 606
initialized() (IPython.parallel.apps.ipclusterapp.IPClusterStop
 class method), 611
initialized() (IPython.parallel.apps.ipcontrollerapp.IPClusterStop
 class method), 617
initialized() (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 624
initialized() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 method), 632
input_hist_parsed (IPython.core.history.HistoryManager
 attribute), 361
input_hist_raw (IPython.core.history.HistoryManager
 attribute), 361
input_mode (IPython.core.inputsplitter.InputSplitter
 attribute), 371
input_mode (IPython.core.inputsplitter.IPythonInputSplitter
 attribute), 370
input_prefilter() (in module IPython.core.hooks),
 367
InputHookManager (class in IPython.lib.inpthook),
 570
InputSplitter (class in IPython.core.inputsplitter),
 370
InputTermColors (class in IPython.utils.coloransi),
 856
insert (IPython.utils.text.SList attribute), 896
inspect_error() (in module IPython.core.ultratb), 547
instance() (IPython.config.configurable.SingletonConfigurableObject)
 (class in IPython.utils.generics),
 863
Inspector (class in IPython.core.oinspect), 449
 install_payload_page() (in module IPython.core.payloadpage), 456
InteractiveShell (IPython.core.payloadpage), 456
Instance (class in IPython.utils.traitslets), 920
instance() (IPython.config.application.Application
 class method), 265
instance() (IPython.config.configurable.SingletonConfigurable
 class method), 274
instance() (IPython.core.application.BaseIPythonApplication
 class method), 289
instance() (IPython.core.interactiveshell.InteractiveShell
 class method), 383
ProfileApp (IPython.core.profileapp.ProfileApp class

method), 514
`instance()` (IPython.core.profileapp.ProfileCreate class method), 518
`instance()` (IPython.core.profileapp.ProfileList class method), 522
`instance()` (IPython.parallel.apps.baseapp.BaseParallelApplication class method), 589
`instance()` (IPython.parallel.apps.ipclusterapp.IPClusterApp class method), 595
`instance()` (IPython.parallel.apps.ipclusterapp.IPClusterEstimator class method), 600
`instance()` (IPython.parallel.apps.ipclusterapp.IPClusterInstance class method), 606
`instance()` (IPython.parallel.apps.ipclusterapp.IPClusterScheduler class method), 611
`instance()` (IPython.parallel.apps.ipcontrollerapp.IPControllerApit) (IPython.utils.traits.List method), 617
`instance()` (IPython.parallel.apps.ipengineapp.IPEngineApp class method), 624
`instance()` (IPython.parallel.apps.iploggerapp.IPLLoggerApp class method), 632
`instance_init()` (IPython.core.interactiveshell.SeparateUnicode instance method), 416
`instance_init()` (IPython.utils.traits.Any method), 901
`instance_init()` (IPython.utils.traits.Bool method), 902
`instance_init()` (IPython.utils.traits.Bytes method), 903
`instance_init()` (IPython.utils.traits.CaselessStrEnum instance method), 911
`instance_init()` (IPython.utils.traits.CBool method), 904
`instance_init()` (IPython.utils.traits.CBytes method), 905
`instance_init()` (IPython.utils.traits.CComplex method), 906
`instance_init()` (IPython.utils.traits.CFloat method), 907
`instance_init()` (IPython.utils.traits.CInt method), 908
`instance_init()` (IPython.utils.traits.ClassBasedTraitType method), 912
`instance_init()` (IPython.utils.traits.CLong method), 909
`instance_init()` (IPython.utils.traits.Complex method), 913
`instance_init()` (IPython.utils.traits.Container method), 914
`instance_init()` (IPython.utils.traits.CUnicode method), 910
`instance_init()` (IPython.utils.traits.Dict method), 916
`ApplicationInit()` (IPython.utils.traits.DottedObjectName method), 916
`Apponce_init()` (IPython.utils.traits.Enum method), 917
`Estimator_init()` (IPython.utils.traits.Float method), 918
`Instance_init()` (IPython.utils.traits.Instance method), 921
`Stoponce_init()` (IPython.utils.traits.Int method), 921
`InstancemethodApit) (IPython.utils.traits.List method), 923
Apponce_init() (IPython.utils.traits.Long method), 924
Apponce_init() (IPython.utils.traits.ObjectName method), 925
Unicode_init() (IPython.utils.traits.Set method), 927
Instance_init() (IPython.utils.traits.TCPAddress method), 928
Instance_init() (IPython.utils.traits.This method), 929
Instance_init() (IPython.utils.traits.TraitType method), 930
Instance_init() (IPython.utils.traits.Tuple method), 932
Instance_init() (IPython.utils.traits.Type method), 933
Instance_init() (IPython.utils.traits.Unicode method), 934
Int (class in IPython.utils.traits), 921
int_id (IPython.parallel.engine.streamkernel.Kernel attribute), 810
integer_loglevel() (in module IPython.parallel.util), 828
interaction() (IPython.core.debugger.Pdb method), 311
interactive() (in module IPython.parallel.util), 828
InteractiveRunner (class in IPython.lib.irunner), 575
InteractiveShell (class in IPython.core.interactiveshell), 375
InteractiveShellABC (class in IPython.core.interactiveshell), 415`

InteractiveShellApp (class in IPython.core.shellapp), 816
 533
interrupt_kernel() (IPython.parallel.engine.kernelstarter.KernelStarter method), 806
interrupt_then_kill()
 (IPython.parallel.apps.launcher.IPClusterLauncher, 817
 method), 643
interrupt_then_kill()
 (IPython.parallel.apps.launcher.LocalController, 817
 method), 657
interrupt_then_kill()
 (IPython.parallel.apps.launcher.LocalEngine, 817
 method), 660
interrupt_then_kill()
 (IPython.parallel.apps.launcher.LocalEngineSet, 817
 method), 662
interrupt_then_kill()
 (IPython.parallel.apps.launcher.LocalProcessLauncher, 824
 method), 665
interrupt_then_kill()
 (IPython.parallel.apps.launcher.MPIExecConfig, 817
 method), 668
interrupt_then_kill()
 (IPython.parallel.apps.launcher.MPIExecLauncher, 824
 method), 671
interrupt_then_kill()
 (IPython.parallel.apps.launcher.MPIExecLauncher, 824
 method), 674
interrupt_then_kill()
 (IPython.parallel.apps.launcher.SSHController, 817
 method), 698
interrupt_then_kill()
 (IPython.parallel.apps.launcher.SSHEngineLauncher, 817
 method), 701
interrupt_then_kill()
 (IPython.parallel.apps.launcher.SSHEngineSet, 817
 method), 704
interrupt_then_kill()
 (IPython.parallel.apps.launcher.SSHLauncher, 817
 method), 706
intersection (IPython.parallel.controller.dependency.Dependency, 615
 attribute), 773
intersection_update (IPython.parallel.controller.dependency.Dependency, 615
 attribute), 773
intro (IPython.core.debugger.Pdb attribute), 312
InvalidAliasError (class in IPython.core.alias), 286
InvalidClientID (class in IPython.parallel.error), 816
InvalidDeferredID (class in IPython.parallel.error), 817
 ip (IPython.parallel.controller.hub.HubFactory attribute), 792
 iopub (IPython.parallel.controller.hub.HubFactory attribute), 810
 IOStream (class in IPython.utils.io), 865
 IOTracker (class in IPython.utils.io), 865
 ip (IPython.parallel.controller.hub.HubFactory attribute), 792
 IPython.parallel.engine.EngineFactory attribute), 804
 ip (IPython.parallel.factory.RegistrationFactory attribute), 824
 ipcluster_args (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 643
 ipcluster_n (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 643
 ipcluster_subcommand (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 643
 IPClusterApp (class in IPython.parallel.apps.ipclusterapp), 594
 IPClusterEngines (class in IPython.parallel.apps.ipclusterapp), 597
 IPClusterLauncher (class in IPython.parallel.apps.ipclusterapp), 603
 IPClusterStart (class in IPython.parallel.apps.ipclusterapp), 609
 IPCompleter (class in IPython.core.completer), 300
 IPControllerApp (class in IPython.parallel.apps.ipcontrollerapp), 721
 IPControllerJob (class in IPython.parallel.apps.winhpcjob), 725
 IPControllerTask (class in IPython.parallel.apps.winhpcjob), 725
 ipdocstring() (in module IPython.testing.ipunitest), 838
 IPEngineApp (class in IPython.parallel.error), 816
 in

| | | |
|--|----|--|
| IPython.parallel.apps.ipengineapp), 622 | | IPython.core.interactiveshell (module), 375 |
| IPEngineSetJob (class in IPython.parallel.apps.winhpcjob), 727 | in | IPython.core.ipapi (module), 417 |
| IPEngineTask (class in IPython.parallel.apps.winhpcjob), 730 | in | IPython.core.logger (module), 417 |
| ipexec() (in module IPython.testing.tools), 847 | | IPython.core.macro (module), 419 |
| ipexec_validate() (in module IPython.testing.tools), 847 | | IPython.core.magic (module), 420 |
| ipfunc() (in module IPython.testing.plugin.dtexample), 840 | | IPython.core.magic_arguments (module), 444 |
| IPLLoggerApp (class in IPython.parallel.apps.iploggerapp), 630 | in | IPython.core.oinspect (module), 448 |
| ipnsdict (class in IPython.testing.globalipapp), 833 | | IPython.core.page (module), 452 |
| iprand() (in module IPython.testing.plugin.dtexample), 841 | | IPython.core.payload (module), 453 |
| iprand_all() (in module IPython.testing.plugin.dtexample), 841 | | IPython.core.payloadpage (module), 456 |
| IPTester (class in IPython.testing.iptest), 835 | | IPython.core.plugin (module), 457 |
| IPyAutocall (class in IPython.core.autocall), 293 | | IPython.core.prefilter (module), 461 |
| IPyAutocallChecker (class in IPython.core.prefilter), 484 | | IPython.core.profileapp (module), 512 |
| ipyfunc2() (in module IPython.testing.plugin.simple), 843 | | IPython.core.profiledir (module), 525 |
| IPyPromptTransformer (class in IPython.core.prefilter), 485 | in | IPython.core.prompts (module), 529 |
| IPython.config.application (module), 263 | | IPython.core.shellapp (module), 532 |
| IPython.config.configurable (module), 268 | | IPython.core.splitinput (module), 535 |
| IPython.config.loader (module), 276 | | IPython.core.ultrathb (module), 536 |
| IPython.core.alias (module), 283 | | IPython.lib.backgroundjobs (module), 547 |
| IPython.core.application (module), 287 | | IPython.lib.clipboard (module), 553 |
| IPython.core.autocall (module), 292 | | IPython.lib.deepcopy (module), 553 |
| IPython.core.builtin_trap (module), 294 | | IPython.lib.demo (module), 554 |
| IPython.core.compilerop (module), 296 | | IPython.lib.guisupport (module), 568 |
| IPython.core.completer (module), 297 | | IPython.lib.inpthook (module), 570 |
| IPython.core.completerlib (module), 303 | | IPython.lib.irunner (module), 573 |
| IPython.core.crashhandler (module), 304 | | IPython.lib.latextools (module), 579 |
| IPython.core.debugger (module), 306 | | IPython.lib.pretty (module), 580 |
| IPython.core.display (module), 315 | | IPython.lib.pylabtools (module), 585 |
| IPython.core.display_trap (module), 316 | | IPython.parallel.apps.baseapp (module), 587 |
| IPython.core.displayhook (module), 318 | | IPython.parallel.apps.ipclusterapp (module), 593 |
| IPython.core.displaypub (module), 322 | | IPython.parallel.apps.ipcontrollerapp (module), 615 |
| IPython.core.error (module), 326 | | IPython.parallel.apps.ipengineapp (module), 621 |
| IPython.core.excolors (module), 327 | | IPython.parallel.apps.iploggerapp (module), 630 |
| IPython.core.extensions (module), 328 | | IPython.parallel.apps.launcher (module), 636 |
| IPython.core.formatters (module), 330 | | IPython.parallel.apps.logwatcher (module), 717 |
| IPython.core.history (module), 358 | | IPython.parallel.apps.win32support (module), 720 |
| IPython.core.hooks (module), 366 | | IPython.parallel.apps.winhpcjob (module), 721 |
| IPython.core.inputsplitter (module), 368 | | IPython.parallel.client.asyncresult (module), 739 |
| | | IPython.parallel.client.client (module), 743 |
| | | IPython.parallel.client.map (module), 752 |
| | | IPython.parallel.client.remotefunction (module), 754 |
| | | IPython.parallel.client.view (module), 756 |
| | | IPython.parallel.controller.dependency (module), 772 |
| | | IPython.parallel.controller.dictdb (module), 775 |
| | | IPython.parallel.controller.heartmonitor (module), 781 |

IPython.parallel.controller.hub (module), 784
IPython.parallel.controller.scheduler (module), 794
IPython.parallel.controller.sqlitedb (module), 800
IPython.parallel.engine.engine (module), 803
IPython.parallel.engine.kernelstarter (module), 806
IPython.parallel.engine.streamkernel (module), 808
IPython.parallel.error (module), 812
IPython.parallel.factory (module), 823
IPython.parallel.util (module), 826
IPython.testing (module), 829
IPython.testing.decorators (module), 830
IPython.testing.globalipapp (module), 832
IPython.testing.ptest (module), 835
IPython.testing.ipunittest (module), 836
IPython.testing.mkdoctests (module), 838
IPython.testing.nosepatch (module), 840
IPython.testing.plugin.dtexample (module), 840
IPython.testing.plugin.show_refs (module), 842
IPython.testing.plugin.simple (module), 843
IPython.testing.plugin.test_ipdoctest (module), 843
IPython.testing.plugin.test_refs (module), 844
IPython.testing.skipdoctest (module), 845
IPython.testing.tools (module), 845
IPython.utils.attic (module), 850
IPython.utils.autoattr (module), 851
IPython.utils.codeutil (module), 854
IPython.utils.coloransi (module), 854
IPython.utils.daemonize (module), 859
IPython.utils.data (module), 859
IPython.utils.decorators (module), 860
IPython.utils.dir2 (module), 860
IPython.utils.doctestreload (module), 860
IPython.utils.frame (module), 861
IPython.utils.generics (module), 862
IPython.utils.growl (module), 863
IPython.utils.importstring (module), 864
IPython.utils.io (module), 864
IPython.utils.ipstruct (module), 867
IPython.utils.jsonutil (module), 871
IPython.utils.newserialized (module), 873
IPython.utils.notification (module), 875
IPython.utils.path (module), 877
IPython.utils.pickleshare (module), 880
IPython.utils.pickleutil (module), 882
IPython.utils.process (module), 884
IPython.utils.PyColorize (module), 848
IPython.utils.strdispatch (module), 885
IPython.utils.sysinfo (module), 886
IPython.utils.syspathcontext (module), 887
IPython.utils.terminal (module), 888
IPython.utils.text (module), 889
IPython.utils.timing (module), 899
IPython.utils.traits (module), 900
IPython.utils.upgradedir (module), 935
IPython.utils.warn (module), 935
IPython.utils.wildcard (module), 936
IPython2PythonConverter (class in IPython.testing.ipunittest), 837
IPYTHON_DIR, 183
ipython_dir (IPython.core.application.BaseIPythonApplication attribute), 290
ipython_dir (IPython.core.interactiveshell.InteractiveShell attribute), 384
ipython_dir (IPython.core.profileapp.ProfileCreate attribute), 519
ipython_dir (IPython.core.profileapp.ProfileList attribute), 523
ipython_dir (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 590
ipython_dir (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 600
ipython_dir (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 606
ipython_dir (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 612
ipython_dir (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 618
ipython_dir (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 624
ipython_dir (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 633
ipython_extension_dir (IPython.core.extensions.ExtensionManager attribute), 329
IPythonCoreError (class in IPython.core.error), 326
IPythonDemo (class in IPython.lib.demo), 563
IPythonError (class in IPython.parallel.error), 815
IPythonGrowlError (class in IPython.utils.growl), 863
IPythonInputSplitter (class in IPython.core.inputsplitter), 369
IPythonLineDemo (class in IPython.lib.demo), 564
IPythonRunner (class in IPython.lib.irunner), 574
is_alive (IPython.parallel.engine.kernelstarter.KernelStarter attribute), 807
is_alive() (IPython.core.history.HistorySavingThread

method), 363
`is_alive()` (IPython.lib.backgroundjobs.BackgroundJobBase method), 363
 method), 548
`is_alive()` (IPython.lib.backgroundjobs.BackgroundJobExpr method), 548
 method), 549
`is_alive()` (IPython.lib.backgroundjobs.BackgroundJobFunc method), 549
 method), 550
`is_alive()` (IPython.parallel.apps.win32support.ForwarderThread method), 550
 method), 720
`is_event_loop_running_qt4()` (in module IPython.lib.guisupport), 569
`is_event_loop_running_wx()` (in module IPython.lib.guisupport), 569
`is_exclusive` (IPython.parallel.apps.winhpcjob.IPController, class in IPython.utils.newserialized), attribute), 723
`is_exclusive` (IPython.parallel.apps.winhpcjob.IPEngine, class in IPython.utils.oldserialized), attribute), 728
`is_exclusive` (IPython.parallel.apps.winhpcjob.WinHPGJob, class in IPython.utils.oldserialized), attribute), 734
`is_importable()` (in module IPython.core.completerlib), 303
`is_parametric` (IPython.parallel.apps.winhpcjob.IPController, class in IPython.parallel.controller.dependency), attribute), 726
`is_parametric` (IPython.parallel.apps.winhpcjob.IPEngine, class in IPython.parallel.controller.dependency), attribute), 731
`is_parametric` (IPython.parallel.apps.winhpcjob.WinHPCJob, class in IPython.utils.text), attribute), 737
`is_rerunnable` (IPython.parallel.apps.winhpcjob.IPController, class in IPython.config.loader.Config), attribute), 726
`is_rerunnable` (IPython.parallel.apps.winhpcjob.IPEngineTask, class in IPython.parallel.controller.dependency), attribute), 731
`is_rerunnable` (IPython.parallel.apps.winhpcjob.WinHPCTask, class in IPython.utils.text), attribute), 737
`is_shadowed()` (in module IPython.core.prefilter), 512
`is_type()` (in module IPython.utils.wildcard), 936
`isAlive()` (IPython.core.history.HistorySavingThread method), 363
`isAlive()` (IPython.lib.backgroundjobs.BackgroundJobBase, class in IPython.utils.ipstruct.Struct), 869
 method), 548
`isAlive()` (IPython.lib.backgroundjobs.BackgroundJobExpr method), 881
 method), 549
`isAlive()` (IPython.lib.backgroundjobs.BackgroundJobFunc method), 279
`isAlive()` (IPython.parallel.apps.win32support.ForwarderThread method), 751
 method), 720
`isalnum` (IPython.utils.text.LSString attribute), 892
`isalpha` (IPython.utils.text.LSString attribute), 892
`isDaemon()` (IPython.core.history.HistorySavingThread method), 363
`isDaemon()` (IPython.lib.backgroundjobs.BackgroundJobBase method), 548
`isDaemon()` (IPython.lib.backgroundjobs.BackgroundJobExpr method), 549
`isDaemon()` (IPython.lib.backgroundjobs.BackgroundJobFunc method), 549
`isDaemon()` (IPython.parallel.apps.win32support.ForwarderThread method), 720
 module), 720
`isdigit` (IPython.utils.text.LSString attribute), 892
`isdisjoint` (IPython.parallel.controller.dependency.Dependency attribute), 773
`Serialized` (IPython.utils.oldserialized, class in IPython.utils.oldserialized), 873
`SetIdentifier` (IPython.utils.traits.DottedObjectName method), 917
`Identifier` (IPython.utils.traits.ObjectName method), 926
`islower` (IPython.utils.text.LSString attribute), 892
`isspace` (IPython.utils.text.LSString attribute), 892
`isupper` (IPython.utils.text.LSString attribute), 892
`items` (IPython.parallel.client.client.Metadata attribute), 773
`items` (IPython.parallel.util.Namespace attribute), 826
`items` (IPython.parallel.util.ReverseDict attribute), 827
`items` (IPython.testing.globalipapp.ipnsdict attribute), 833
`items` (IPython.utils.coloransi.ColorSchemeTable attribute), 855
`items` (IPython.utils.ipstruct.Struct attribute), 869
`items()` (IPython.utils.pickleshare.PickleShareDB method), 881
`iteritems` (IPython.config.loader.Config attribute), 279
`iteritems` (IPython.parallel.client.client.Metadata attribute), 751
`iteritems` (IPython.parallel.util.Namespace attribute), 826
`iteritems` (IPython.parallel.util.ReverseDict attribute), 827

tribute), 827
iteritems (IPython.testing.globalipapp.ipnsdict attribute), 834
iteritems (IPython.utils.coloransi.ColorSchemeTable attribute), 855
iteritems (IPython.utils.ipstruct.Struct attribute), 869
iteritems() (IPython.utils.pickleshare.PickleShareDB method), 881
iterkeys (IPython.config.loader.Config attribute), 279
iterkeys (IPython.parallel.client.client.Metadata attribute), 751
iterkeys (IPython.parallel.util.Namespace attribute), 826
iterkeys (IPython.parallel.util.ReverseDict attribute), 827
iterkeys (IPython.testing.globalipapp.ipnsdict attribute), 834
iterkeys (IPython.utils.coloransi.ColorSchemeTable attribute), 855
iterkeys (IPython.utils.ipstruct.Struct attribute), 869
iterkeys() (IPython.utils.pickleshare.PickleShareDB method), 881
itervalues (IPython.config.loader.Config attribute), 279
itervalues (IPython.parallel.client.client.Metadata attribute), 752
itervalues (IPython.parallel.util.Namespace attribute), 826
itervalues (IPython.parallel.util.ReverseDict attribute), 828
itervalues (IPython.testing.globalipapp.ipnsdict attribute), 834
itervalues (IPython.utils.coloransi.ColorSchemeTable attribute), 856
itervalues (IPython.utils.ipstruct.Struct attribute), 869
itervalues() (IPython.utils.pickleshare.PickleShareDB method), 881
IUnSerialized (class in IPython.utils.newserialized), 873

J

JavascriptFormatter (class in IPython.core.formatters), 342

job_array_regex (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 640

job_array_regex (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 646

job_array_regex (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 650

job_array_regex (IPython.parallel.apps.launcher.LSFLauncher attribute), 653

job_array_regex (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 677

job_array_regex (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 680

job_array_regex (IPython.parallel.apps.launcher.PBSLauncher attribute), 684

job_array_regex (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 688

job_array_regex (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 691

job_array_regex (IPython.parallel.apps.launcher.SGELauncher attribute), 695

job_array_template (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 640

job_array_template (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 646

job_array_template (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 650

job_array_template (IPython.parallel.apps.launcher.LSFLauncher attribute), 653

job_array_template (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 677

job_array_template (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 680

job_array_template (IPython.parallel.apps.launcher.PBSLauncher attribute), 684

job_array_template (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 688

job_array_template (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 691

job_array_template (IPython.parallel.apps.launcher.SGELauncher attribute), 695

job_cmd (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher attribute), 710

job_cmd (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher attribute), 713

job_cmd (IPython.parallel.apps.launcher.WindowsHPCLauncher attribute), 715

job_file (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher attribute), 710

job_file (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher attribute), 713

keys (IPython.testing.globalipapp.ipnsdict attribute), 834
keys (IPython.utils.coloransi.ColorSchemeTable attribute), 856
keys (IPython.utils.ipstruct.Struct attribute), 869
keys() (IPython.utils.pickleshare.PickleShareDB method), 881
keytable (IPython.parallel.controller.hub.Hub attribute), 789
keyvalue_description (IPython.config.application.Application attribute), 265
keyvalue_description (IPython.core.application.BaseIPythonApplication attribute), 290
keyvalue_description (IPython.core.profileapp.ProfileApp attribute), 514
keyvalue_description (IPython.core.profileapp.ProfileCreate attribute), 519
keyvalue_description (IPython.core.profileapp.ProfileList attribute), 523
keyvalue_description (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 590
keyvalue_description (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 595
keyvalue_description (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 600
keyvalue_description (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 606
keyvalue_description (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 612
keyvalue_description (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 618
keyvalue_description (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 624
keyvalue_description (IPython.parallel.apps.iploggerapp.IPLoggerApp attribute), 633
KeyValueConfigLoader (class in IPython.config.loader), 281
kill() (IPython.parallel.client.view.DirectView method), 758
kill_kernel() (IPython.parallel.engine.kernelstarter.KernelStarter method), 807
klass (IPython.utils.traits.Container attribute), 915
klass (IPython.utils.traits.List attribute), 923
klass (IPython.utils.traits.Set attribute), 927
klass (IPython.utils.traits.Tuple attribute), 932
kwds (class in IPython.core.magic_arguments), 447

L

l (IPython.utils.text.LSString attribute), 892
last_ping (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 782
lastcmd (IPython.core.debugger.Pdb attribute), 312
late_startup_hook() (in module IPython.core.hooks), 367
latex_to_html() (in module IPython.lib.latextools), 579
latex_to_png() (in module IPython.lib.latextools), 579
LatexFormatter (class in IPython.core.formatters), 345
launch_new_instance() (in module IPython.parallel.apps.ipclusterapp), 614
launch_new_instance() (in module IPython.parallel.apps.ipcontrollerapp), 621
Engines new_instance() (in module IPython.parallel.apps.ipengineapp), 629
launch_new_instance() (in module IPython.parallel.apps.iploggerapp), 635
launch_scheduler() (in module IPython.parallel.controller.scheduler), 799
launcher_class (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 662
SSHEngineSetLauncherApp class (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 704
LauncherError (class in IPython.parallel.apps.launcher), 656
launchers (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 662
SSHEngineSetLauncherApp launchers (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 704

leastload() (in module IPython.parallel.controller.scheduler), 799
length_error() (IPython.utils.traits.List method), 923
lifetime (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 782
LightBlue (IPython.utils.coloransi.InputTermColors attribute), 857
LightBlue (IPython.utils.coloransi.TermColors attribute), 858
LightCyan (IPython.utils.coloransi.InputTermColors attribute), 857
LightCyan (IPython.utils.coloransi.TermColors attribute), 858
LightGray (IPython.utils.coloransi.InputTermColors attribute), 857
LightGray (IPython.utils.coloransi.TermColors attribute), 858
LightGreen (IPython.utils.coloransi.InputTermColors attribute), 857
LightGreen (IPython.utils.coloransi.TermColors attribute), 858
LightPurple (IPython.utils.coloransi.InputTermColors attribute), 857
LightPurple (IPython.utils.coloransi.TermColors attribute), 858
LightRed (IPython.utils.coloransi.InputTermColors attribute), 857
LightRed (IPython.utils.coloransi.TermColors attribute), 858
LineDemo (class in IPython.lib.demo), 566
LineInfo (class in IPython.core.inputsplitter), 372
LineInfo (class in IPython.core.prefilter), 487
lineinfo() (IPython.core.debugger.Pdb method), 312
List (class in IPython.utils.traits), 922
list (IPython.utils.text.LSString attribute), 892
list (IPython.utils.text.SList attribute), 896
list2dict() (in module IPython.utils.data), 859
list2dict2() (in module IPython.utils.data), 859
list_command_pydb() (IPython.core.debugger.Pdb method), 312
list_namespace() (in module IPython.utils.wildcard), 936
list_profile_dirs() (IPython.core.profileapp.ProfileList method), 523
list_strings() (in module IPython.utils.text), 898
ListTB (class in IPython.core.ultratb), 542
ljust (IPython.utils.text.LSString attribute), 892
load_balanced_view() (IPython.parallel.client.Client method), 748
load_config() (IPython.config.loader.ArgParseConfigLoader method), 277
load_config() (IPython.config.loader.CommandLineConfigLoader method), 278
load_config() (IPython.config.loader.ConfigLoader method), 280
load_config() (IPython.config.loader.FileConfigLoader method), 281
load_config() (IPython.config.loader.KeyValueConfigLoader method), 282
load_config() (IPython.config.loader.PyFileConfigLoader method), 283
load_config_file() (IPython.config.application.Application method), 265
load_config_file() (IPython.core.application.BaseIPythonApplication method), 290
load_config_file() (IPython.core.profileapp.ProfileApp method), 515
load_config_file() (IPython.core.profileapp.ProfileCreate method), 519
load_config_file() (IPython.core.profileapp.ProfileList method), 523
load_config_file() (IPython.parallel.apps.baseapp.BaseParallelApp method), 590
load_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterApp method), 596
load_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 600
load_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 606
load_config_file() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 612
load_config_file() (IPython.parallel.apps.ipcontrollerapp.IPCController method), 618
load_config_file() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 625
load_config_file() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 633
load_config_from_json() (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp method), 618
load_extension() (IPython.core.extensions.ExtensionManager method), 329
load_tail() (in module IPython.lib.deepreload), 554

LoadBalancedView (class in IPython.parallel.client.view), 762
loads (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
local_logger() (in module IPython.parallel.util), 828
LocalControllerLauncher (class in IPython.parallel.apps.launcher), 656
LocalEngineLauncher (class in IPython.parallel.apps.launcher), 659
LocalEngineSetLauncher (class in IPython.parallel.apps.launcher), 661
LocalProcessLauncher (class in IPython.parallel.apps.launcher), 664
location (IPython.core.profiledir.ProfileDir attribute), 528
location (IPython.parallel.apps.ipcontrollerapp.IPCONTROLLER attribute), 618
location (IPython.parallel.apps.launcher.SSHController attribute), 698
location (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 701
location (IPython.parallel.apps.launcher.SSHLauncher log (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 706
location (IPython.parallel.controller.sqlite3.SQLiteDB attribute), 801
location (IPython.parallel.engine.EngineFactory log (IPython.parallel.apps.launcher.SSHLauncher attribute), 804
log (IPython.config.configurable.LoggingConfigurable log (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher attribute), 272
log (IPython.parallel.apps.launcher.BaseLauncher attribute), 637
log (IPython.parallel.apps.launcher.BatchSystemLauncher log (IPython.parallel.apps.launcher.WindowsHPCLauncher attribute), 640
log (IPython.parallel.apps.launcher.IPClusterLauncher log (IPython.parallel.apps.logwatcher.LogWatcher attribute), 643
log (IPython.parallel.apps.launcher.LocalControllerLauncher log (IPython.parallel.controller.dictdb.BaseDB attribute), 657
log (IPython.parallel.apps.launcher.LocalEngineLauncher log (IPython.parallel.controller.dictdb.DictDB attribute), 660
log (IPython.parallel.apps.launcher.LocalEngineSetLauncher log (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 662
log (IPython.parallel.apps.launcher.LocalProcessLauncher log (IPython.parallel.controller.hub.Hub attribute), 665
log (IPython.parallel.apps.launcher.LSFControllerLauncher log (IPython.parallel.controller.hub.HubFactory attribute), 647
log (IPython.parallel.apps.launcher.LSFEngineSetLauncher log (IPython.parallel.controller.scheduler.TaskScheduler attribute), 650
log (IPython.parallel.apps.launcher.LSFLauncher attribute), 797
log (IPython.parallel.controller.sqlitedb.SQLiteDB

attribute), 802
`log` (IPython.parallel.engine.EngineFactory attribute), 804
`log` (IPython.parallel.engine.streamkernel.Kernel attribute), 810
`log` (IPython.parallel.factory.RegistrationFactory attribute), 824
`log()` (IPython.core.logger.Logger method), 418
`log_dir` (IPython.core.profiledir.ProfileDir attribute), 528
`log_dir_name` (IPython.core.profiledir.ProfileDir attribute), 528
`log_level` (IPython.config.application.Application attribute), 265
`log_level` (IPython.core.application.BaseIPythonApplication attribute), 290
`log_level` (IPython.core.profileapp.ProfileApp attribute), 515
`log_level` (IPython.core.profileapp.ProfileCreate attribute), 519
`log_level` (IPython.core.profileapp.ProfileList attribute), 523
`log_level` (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 590
`log_level` (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 596
`log_level` (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 600
`log_level` (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 625
`log_to_file` (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 633
`log_url` (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 590
`log_url` (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 600
`log_url` (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 606
`log_url` (IPython.parallel.apps.ipcontrollerapp.IPClusterStop attribute), 612
`log_url` (IPython.parallel.apps.ipcontrollerapp.IPClusterStop attribute), 618
`log_url` (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 625
`log_url` (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 633
`log_url` (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 633
`log_url` (IPython.parallel.apps.ipcontrollerapp.IPClusterStop attribute), 618
`log_append` (IPython.core.interactiveshell.InteractiveShell attribute), 384
`log_file` (IPython.core.interactiveshell.InteractiveShell attribute), 384
`log_started()` (in module IPython.parallel.controller.scheduler), 799
`Logger` (class in IPython.core.logger), 418
`LoggerConfigurable` (class in IPython.config.configurable), 270
`logmode` (IPython.core.logger.Logger attribute), 418
`logname` (IPython.parallel.controller.hub.Hub attribute), 789
`logname` (IPython.parallel.controller.hub.HubFactory attribute), 793
`logname` (IPython.parallel.controller.scheduler.TaskScheduler attribute), 797
`logname` (IPython.parallel.engine.EngineFactory attribute), 804
`logname` (IPython.parallel.engine.streamkernel.Kernel attribute), 810
`logname` (IPython.parallel.factory.RegistrationFactory attribute), 824
`logstart` (IPython.core.interactiveshell.InteractiveShell attribute), 606

attribute), 384
logstart() (IPython.core.logger.Logger method), 418
logstate() (IPython.core.logger.Logger method), 418
logstop() (IPython.core.logger.Logger method), 418
LogWatcher (class in IPython.parallel.apps.logwatcher), 718
Long (class in IPython.utils.traits), 924
lookupmodule() (IPython.core.debugger.Pdb method), 312
loop (IPython.parallel.apps.baseapp.BaseParallelApplication), 590
loop (IPython.parallel.apps.ipclusterapp.IPClusterEngine), 601
loop (IPython.parallel.apps.ipclusterapp.IPClusterStartloop), 606
loop (IPython.parallel.apps.ipclusterapp.IPClusterStoploop), 612
loop (IPython.parallel.apps.ipcontrollerapp.IPControllerApp), 618
loop (IPython.parallel.apps.ipengineapp.IPEngineApp), 625
loop (IPython.parallel.apps.iploggerapp.IPLoaderApp), 633
loop (IPython.parallel.apps.launcher.BaseLauncher), 637
loop (IPython.parallel.apps.launcher.BatchSystemLauncher), 640
loop (IPython.parallel.apps.launcher.IPClusterLauncher), 644
loop (IPython.parallel.apps.launcher.LocalControllerLauncher), 657
loop (IPython.parallel.apps.launcher.LocalEngineLauncher), 660
loop (IPython.parallel.apps.launcher.LocalEngineSetLauncher), 663
loop (IPython.parallel.apps.launcher.LocalProcessLauncher), 665
loop (IPython.parallel.apps.launcher.LSFControllerLauncher), 647
loop (IPython.parallel.apps.launcher.LSFEngineSetLauncher), 650
loop (IPython.parallel.apps.launcher.LSFLauncher), 654
loop (IPython.parallel.apps.launcher.MPIExecController), 668
loop (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher), 671
loop (IPython.parallel.apps.launcher.MPIExecLauncher), 674
loop (IPython.parallel.apps.launcher.PBSControllerLauncher), 677
loop (IPython.parallel.apps.launcher.PBSEngineSetLauncher), 681
loop (IPython.parallel.apps.launcher.PBSLauncher), 684
loop (IPython.parallel.apps.launcher.SGEControllerLauncher), 688
loop (IPython.parallel.apps.launcher.SGEEngineSetLauncher), 691
loop (IPython.parallel.apps.launcher.SGELauncher), 695
loop (IPython.parallel.apps.launcher.SSHControllerLauncher), 698
loop (IPython.parallel.apps.launcher.SSHEngineLauncher), 701
loop (IPython.parallel.apps.launcher.SSHEngineSetLauncher), 704
loop (IPython.parallel.apps.launcher.SSHLauncher), 707
loop (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher), 710
loop (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher), 713
loop (IPython.parallel.apps.launcher.WindowsHPCLauncher), 716
loop (IPython.parallel.apps.logwatcher.LogWatcher), 719
loop (IPython.parallel.controller.HeartMonitor), 783
loop (IPython.parallel.controller.hub.Hub), 789
loop (IPython.parallel.controller.hub.HubFactory), 793
loop (IPython.parallel.controller.scheduler.TaskScheduler), 797
loop (IPython.parallel.engine.EngineFactory), 804
loop (IPython.parallel.engine.streamkernel.Kernel), 810
loop (IPython.parallel.factory.RegistrationFactory), 824
loop (IPython.utils.text.LSString), 893
lru() (in module IPython.parallel.controller.scheduler), 799
LSFControllerLauncher (class in IPython.parallel.apps.launcher), 645

LSFEngineSetLauncher (class in IPython.parallel.apps.launcher), 649

LSFLauncher (class in IPython.parallel.apps.launcher), 652

lsmagic() (IPython.core.interactiveshell.InteractiveShell method), 384

lsmagic() (IPython.core.magic.Magic method), 421

LSString (class in IPython.utils.text), 890

lstrip (IPython.utils.text.LSString attribute), 893

M

Macro (class in IPython.core.macro), 419

MacroChecker (class in IPython.core.prefilter), 488

MacroHandler (class in IPython.core.prefilter), 490

MacroToEdit (class in IPython.core.magic), 420

Magic (class in IPython.core.magic), 420

magic() (IPython.core.interactiveshell.InteractiveShell method), 384

magic_alias() (IPython.core.interactiveshell.InteractiveShell method), 385

magic_alias() (IPython.core.magic.Magic method), 421

magic_arguments (class in IPython.core.magic_arguments), 447

magic_autocall() (IPython.core.interactiveshell.InteractiveShell method), 385

magic_autocall() (IPython.core.magic.Magic method), 422

magic_automagic() (IPython.core.interactiveshell.InteractiveShell method), 386

magic_automagic() (IPython.core.magic.Magic method), 422

magic_bookmark() (IPython.core.interactiveshell.InteractiveShell method), 386

magic_bookmark() (IPython.core.magic.Magic method), 423

magic_cd() (IPython.core.interactiveshell.InteractiveShell method), 386

magic_cd() (IPython.core.magic.Magic method), 423

magic_colors() (IPython.core.interactiveshell.InteractiveShell method), 387

magic_colors() (IPython.core.magic.Magic method), 424

magic_debug() (IPython.core.interactiveshell.InteractiveShell method), 387

magic_debug() (IPython.core.magic.Magic method), 424

in magic_dhist() (IPython.core.interactiveshell.InteractiveShell method), 387

in magic_dhist() (IPython.core.magic.Magic method), 424

magic_dirs() (IPython.core.interactiveshell.InteractiveShell method), 388

magic_dirs() (IPython.core.magic.Magic method), 424

magic_doctest_mode()

(IPython.core.interactiveshell.InteractiveShell method), 388

magic_doctest_mode() (IPython.core.magic.Magic method), 424

magic_ed() (IPython.core.interactiveshell.InteractiveShell method), 388

magic_ed() (IPython.core.magic.Magic method), 425

magic_edit() (IPython.core.interactiveshell.InteractiveShell method), 388

magic_edit() (IPython.core.magic.Magic method), 425

magic_env() (IPython.core.interactiveshell.InteractiveShell method), 390

magic_env() (IPython.core.magic.Magic method), 427

magic_gui() (IPython.core.interactiveshell.InteractiveShell method), 390

magic_gui() (IPython.core.magic.Magic method), 427

magic_history() (in module IPython.core.history), 364

magic_install_default_config()

(IPython.core.interactiveshell.InteractiveShell method), 390

magic_install_profiles()

(IPython.core.interactiveshell.InteractiveShell method), 391

magic_load_ext() (IPython.core.interactiveshell.InteractiveShell method), 391

magic_load_ext() (IPython.core.magic.Magic method), 427

magic_loadpy() (IPython.core.interactiveshell.InteractiveShell method), 391

magic_loadpy() (IPython.core.magic.Magic method), 427
magic_logoff() (IPython.core.interactiveshell.InteractiveShell method), 391
magic_logoff() (IPython.core.magic.Magic method), magic_pdef() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_logon() (IPython.core.interactiveshell.InteractiveShell method), 428
magic_logon() (IPython.core.interactiveshell.InteractiveShell method), magic_pdef() (IPython.core.magic.Magic method), 430
magic_logon() (IPython.core.magic.Magic method), magic_pdoc() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_logstart() (IPython.core.interactiveshell.InteractiveShell method), 428
magic_logstart() (IPython.core.interactiveshell.InteractiveShell method), magic_pfile() (IPython.core.interactiveshell.InteractiveShell method), 394
magic_logstart() (IPython.core.interactiveshell.InteractiveShell method), 430
magic_logstate() (IPython.core.interactiveshell.InteractiveShell method), 428
magic_logstate() (IPython.core.interactiveshell.InteractiveShell method), magic_pinfo() (IPython.core.interactiveshell.InteractiveShell method), 394
magic_logstop() (IPython.core.interactiveshell.InteractiveShell method), 392
magic_logstop() (IPython.core.interactiveshell.InteractiveShell method), magic_pinfo2() (IPython.core.interactiveshell.InteractiveShell method), 394
magic_logstop() (IPython.core.interactiveshell.InteractiveShell method), 430
magic_logstop() (IPython.core.interactiveshell.InteractiveShell method), magic_popd() (IPython.core.interactiveshell.InteractiveShell method), 394
magic_lsmagic() (IPython.core.interactiveshell.InteractiveShell method), 428
magic_lsmagic() (IPython.core.interactiveshell.InteractiveShell method), magic_pprint() (IPython.core.interactiveshell.InteractiveShell method), 394
magic_macro() (IPython.core.interactiveshell.InteractiveShell method), 392
magic_macro() (IPython.core.interactiveshell.InteractiveShell method), magic_pprint() (IPython.core.interactiveshell.InteractiveShell method), 431
magic_macro() (IPython.core.interactiveshell.InteractiveShell method), 429
magic_precision() (IPython.core.interactiveshell.InteractiveShell method), 429
magic_magic() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_magic() (IPython.core.interactiveshell.InteractiveShell method), magic_precision() (IPython.core.interactiveshell.InteractiveShell method), 431
magic_magic() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_magic() (IPython.core.interactiveshell.InteractiveShell method), magic_profile() (IPython.core.interactiveshell.InteractiveShell method), 395
magic_matches() (IPython.core.completer.ICompleter method), 302
magic_page() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_page() (IPython.core.interactiveshell.InteractiveShell method), magic_prun() (IPython.core.interactiveshell.InteractiveShell method), 429
magic_pastebin() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_pastebin() (IPython.core.interactiveshell.InteractiveShell method), magic_psearch() (IPython.core.interactiveshell.InteractiveShell method), 430
magic_pdb() (IPython.core.interactiveshell.InteractiveShell method), 393
magic_pdb() (IPython.core.interactiveshell.InteractiveShell method), magic_psolve() (IPython.core.interactiveshell.InteractiveShell method), 397
magic_pdb() (IPython.core.interactiveshell.InteractiveShell method), 430
magic_psolve() (IPython.core.interactiveshell.InteractiveShell method), 434

magic_pushd() (IPython.core.interactiveshell.InteractiveShell method), 397
magic_pushd() (IPython.core.magic.Magic method), 434
magic_pwd() (IPython.core.interactiveshell.InteractiveShell method), 397
magic_pwd() (IPython.core.magic.Magic method), 434
magic_pycat() (IPython.core.interactiveshell.InteractiveShell method), 398
magic_pycat() (IPython.core.magic.Magic method), 434
magic_pylab() (IPython.core.interactiveshell.InteractiveShell method), 398
magic_pylab() (IPython.core.magic.Magic method), 434
magic_quickref() (IPython.core.interactiveshell.InteractiveShell method), 398
magic_quickref() (IPython.core.magic.Magic method), 435
magic_rehashx() (IPython.core.interactiveshell.InteractiveShell method), 398
magic_rehashx() (IPython.core.magic.Magic method), 435
magic_reload_ext() (IPython.core.interactiveshell.InteractiveShell method), 398
magic_reload_ext() (IPython.core.magic.Magic method), 435
magic_rep() (in module IPython.core.history), 365
magic_rerun() (in module IPython.core.history), 365
magic_reset() (IPython.core.interactiveshell.InteractiveShell method), 399
magic_reset() (IPython.core.magic.Magic method), 435
magic_reset_selective() (IPython.core.interactiveshell.InteractiveShell method), 399
magic_reset_selective() (IPython.core.magic.Magic method), 435
magic_run() (IPython.core.interactiveshell.InteractiveShell method), 400
magic_run() (IPython.core.magic.Magic method), 436
magic_run_completer() (in module IPython.core.completerlib), 303
magic_save() (IPython.core.interactiveshell.InteractiveShell method), 401
magic_save() (IPython.core.magic.Magic method),

magic_sc() (IPython.core.interactiveshell.InteractiveShell method), 402
magic_sc() (IPython.core.magic.Magic method), 438
magic_sx() (IPython.core.interactiveshell.InteractiveShell method), 403
magic_sx() (IPython.core.magic.Magic method), 438
magic_tb() (IPython.core.interactiveshell.InteractiveShell method), 403
magic_tb() (IPython.core.magic.Magic method), 440
magic_time() (IPython.core.interactiveshell.InteractiveShell method), 403
magic_time() (IPython.core.magic.Magic method), 440
magic_timeit() (IPython.core.interactiveshell.InteractiveShell method), 404
magic_timeit() (IPython.core.magic.Magic method), 441
magic_unalias() (IPython.core.interactiveshell.InteractiveShell method), 405
magic_unalias() (IPython.core.magic.Magic method), 441
magic_unload_ext() (IPython.core.interactiveshell.InteractiveShell method), 405
magic_unload_ext() (IPython.core.magic.Magic method), 441
magic_who() (IPython.core.interactiveshell.InteractiveShell method), 405
magic_who() (IPython.core.magic.Magic method), 441
magic_who_ls() (IPython.core.interactiveshell.InteractiveShell method), 406
magic_who_ls() (IPython.core.magic.Magic method), 442
magic_whos() (IPython.core.interactiveshell.InteractiveShell method), 406
magic_whos() (IPython.core.magic.Magic method), 442
magic_xdel() (IPython.core.interactiveshell.InteractiveShell method), 407
magic_xdel() (IPython.core.magic.Magic method), 443
magic_xmode() (IPython.core.interactiveshell.InteractiveShell method), 407

magic_xmode() (IPython.core.magic.Magic method), 443
MagicArgumentParser (class in IPython.core.magic_arguments), 446
MagicHandler (class in IPython.core.prefilter), 492
main() (in module IPython.lib.iprunner), 578
main() (in module IPython.testing.iptest), 836
main() (in module IPython.testing.mkdoctests), 839
main() (in module IPython.utils.pickleshare), 882
main() (in module IPython.utils.PyColorize), 849
main() (IPython.lib.iprunner.InteractiveRunner method), 575
main() (IPython.lib.iprunner.IPythonRunner method), 574
main() (IPython.lib.iprunner.PythonRunner method), 576
main() (IPython.lib.iprunner.SAGERunner method), 578
make_color_table() (in module IPython.utils.coloransi), 858
make_exclude() (in module IPython.testing.iptest), 836
make_label_dec() (in module IPython.testing.decorators), 830
make_quoted_expr() (in module IPython.utils.text), 898
make_report() (IPython.core.crashhandler.CrashHandler method), 305
make_report() (IPython.parallel.apps.baseapp.ParallelCrashHandler method), 593
make_runners() (in module IPython.testing.iptest), 836
make_starter() (in module IPython.parallel.engine.kernelstarter), 807
make_user_namespaces() (IPython.core.interactiveshell.InteractiveShell max_cores (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 723
Map (class in IPython.parallel.client.map), 753
map() (IPython.parallel.client.remotefunction.ParallelFunction attribute), 729
method), 755
map() (IPython.parallel.client.view.DirectView method), 758
map() (IPython.parallel.client.view.LoadBalancedView method), 764
map() (IPython.parallel.client.view.View method), 769
map_async() (IPython.parallel.client.view.DirectView method), 759
map_async() (IPython.parallel.client.view.LoadBalancedView method), 764
map_async() (IPython.parallel.client.view.View method), 769
map_method() (in module IPython.utils.attic), 850
map_sync() (IPython.parallel.client.view.DirectView method), 759
map_sync() (IPython.parallel.client.view.LoadBalancedView method), 764
map_sync() (IPython.parallel.client.view.View method), 769
mapObject (IPython.parallel.client.remotefunction.ParallelFunction attribute), 755
mappable() (in module IPython.parallel.client.map), 753
mark_dirs() (in module IPython.core.completer), 303
marquee() (in module IPython.utils.text), 898
marquee() (IPython.lib.demo.ClearDemo method), 558
marquee() (IPython.lib.demo.ClearIPDemo method), 559
marquee() (IPython.lib.demo.ClearMixin method), 561
marquee() (IPython.lib.demo.Demo method), 562
marquee() (IPython.lib.demo.IPythonDemo method), 564
max_cores (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 726
max_cores (IPython.parallel.apps.winhpcjob.IPEngineSetJob
attribute), 729
max_cores (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 731
max_cores (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 734
max_cores (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 737
max_nodes (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 723

max_nodes (IPython.parallel.apps.winhpcjob.IPController.message) (IPython.core.interactiveshell.SpaceInInput attribute), 726
max_nodes (IPython.parallel.apps.winhpcjob.IPEngine.message) (IPython.core.magic.MacroToEdit attribute), 420
max_nodes (IPython.parallel.apps.winhpcjob.IPEngine.message) (IPython.core.prefilter.PrefilterError attribute), 497
max_nodes (IPython.parallel.apps.winhpcjob.WinHPCJob.message) (IPython.core.profiledir.ProfileDirError attribute), 529
max_nodes (IPython.parallel.apps.winhpcjob.WinHPCJob.message) (IPython.lib.demo.DemoError attribute), 563
max_sockets (IPython.parallel.apps.winhpcjob.IPController.message) (IPython.parallel.apps.baseapp.PIDFileError attribute), 593
max_sockets (IPython.parallel.apps.winhpcjob.IPController.message) (IPython.parallel.apps.launcher.LauncherError attribute), 656
max_sockets (IPython.parallel.apps.winhpcjob.IPEngine.message) (IPython.parallel.apps.launcher.ProcessStateError attribute), 686
max_sockets (IPython.parallel.apps.winhpcjob.IPEngine.message) (IPython.parallel.apps.launcher.UnknownStatus attribute), 708
max_sockets (IPython.parallel.apps.winhpcjob.WinHPCJob.message) (IPython.parallel.error.AbortedPendingDeferredError attribute), 813
max_sockets (IPython.parallel.apps.winhpcjob.WinHPCJob.message) (IPython.parallel.error.ClientError attribute), 813
max_width (IPython.core.formatters.PlainTextFormatter.message) (IPython.parallel.error.CompositeError attribute), 813
maybe_run() (IPython.parallel.controller.scheduler.TaskScheduler.message) (IPython.parallel.error.ConnectionError attribute), 814
merge() (IPython.utils.ipstruct.Struct method), 869
message (IPython.config.application.ApplicationError.message) (IPython.parallel.error.ControllerCreationError attribute), 814
message (IPython.config.configurable.ConfigurableError.message) (IPython.parallel.error.ControllerError attribute), 814
message (IPython.config.configurable.MultipleInstanceError.message) (IPython.parallel.error.DependencyTimeout attribute), 814
message (IPython.config.loader.ArgumentParser.message) (IPython.parallel.error.EngineCreationError attribute), 815
message (IPython.config.loader.ConfigError.message) (IPython.parallel.error.EngineError attribute), 815
message (IPython.config.loader.ConfigLoaderError.message) (IPython.parallel.error.FileTimeoutError attribute), 815
message (IPython.core.alias.AliasError.message) (IPython.parallel.error.IdInUse attribute), 816
message (IPython.core.alias.InvalidAliasError.message) (IPython.parallel.error.ImpossibleDependency attribute), 816
message (IPython.core.error.IPythonCoreError.message) (IPython.parallel.error.InvalidClientID attribute), 816
message (IPython.core.error.TryNext.message) (IPython.parallel.error.InvalidDeferredID attribute), 816
message (IPython.core.error.UsageError.message) (IPython.parallel.error.InvalidDependency attribute), 817

message (IPython.parallel.error.InvalidEngineID attribute), 817
message (IPython.parallel.error.InvalidProperty attribute), 817
message (IPython.parallel.error.IPythonError attribute), 815
message (IPython.parallel.error.KernelError attribute), 817
message (IPython.parallel.error.MessageSizeError attribute), 818
message (IPython.parallel.error.MissingBlockArgument attribute), 818
message (IPython.parallel.error.NoEnginesRegistered attribute), 818
message (IPython.parallel.error.NotAPendingResult attribute), 818
message (IPython.parallel.error.NotDefined attribute), 819
message (IPython.parallel.error.PBMessageSizeError attribute), 819
message (IPython.parallel.error.ProtocolError attribute), 819
message (IPython.parallel.error.QueueCleared attribute), 819
message (IPython.parallel.error.RemoteError attribute), 820
message (IPython.parallel.error.ResultAlreadyRetrieved attribute), 820
message (IPython.parallel.error.ResultNotCompleted attribute), 820
message (IPython.parallel.error.SecurityError attribute), 821
message (IPython.parallel.error.SerializationError attribute), 821
message (IPython.parallel.error.StopLocalExecution attribute), 821
message (IPython.parallel.error.TaskAborted attribute), 821
message (IPython.parallel.error.TaskRejectError attribute), 822
message (IPython.parallel.error.TaskTimeout attribute), 822
message (IPython.parallel.error.TimeoutError attribute), 822
message (IPython.parallel.error.UnmetDependency attribute), 822
message (IPython.parallel.error.UnpickleableException attribute), 823
message (IPython.utils.growl.IPythonGrowlError attribute), 863
message (IPython.utils.newserialized.SerializationError attribute), 874
message (IPython.utils.notification.NotificationError attribute), 877
message (IPython.utils.path.HomeDirError attribute), 878
message (IPython.utils.process.FindCmdError attribute), 884
message (IPython.utils.traits.TraitError attribute), 929
message_template (IPython.core.crashhandler.CrashHandler attribute), 305
message_template (IPython.parallel.apps.baseapp.ParallelCrashHandler attribute), 593
MessageSizeError (class in IPython.parallel.error), 818
Metadata (class in IPython.parallel.client.client), 751
metadata (IPython.core.interactiveshell.SeparateUnicode attribute), 416
metadata (IPython.parallel.client.asyncresult.AsyncHubResult attribute), 740
metadata (IPython.parallel.client.asyncresult.AsyncMapResult attribute), 741
metadata (IPython.parallel.client.asyncresult.AsyncResult attribute), 742
metadata (IPython.parallel.client.client.Client attribute), 748
metadata (IPython.utils.traits.Any attribute), 901
metadata (IPython.utils.traits.Bool attribute), 902
metadata (IPython.utils.traits.Bytes attribute), 903
metadata (IPython.utils.traits.CaselessStrEnum attribute), 911
metadata (IPython.utils.traits.CBool attribute), 904
metadata (IPython.utils.traits.CBytes attribute), 905
metadata (IPython.utils.traits.CComplex attribute), 906
metadata (IPython.utils.traits.CFloat attribute), 907
metadata (IPython.utils.traits.CInt attribute), 908
metadata (IPython.utils.traits.ClassBasedTraitType attribute), 912
metadata (IPython.utils.traits.CLong attribute), 909
metadata (IPython.utils.traits.Complex attribute),

mpi_args (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 668
mpi_args (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 671
mpi_args (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 674
mpi_cmd (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 668
mpi_cmd (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 671
mpi_cmd (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 674
mpi_cmd (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 893
mpi_cmd (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 896
mpi_cmd (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 265
mpi_cmd (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 290
MPIExecControllerLauncher (class in IPython.core.history.HistorySavingThread attribute), 363
MPIExecEngineSetLauncher (class in IPython.core.profileapp.ProfileApp attribute), 515
MPIExecLauncher (class in IPython.core.profileapp.ProfileCreate attribute), 519
mpl_runner() (in module IPython.lib.pylabtools), name (IPython.core.profileapp.ProfileList attribute), 523
mq_class (IPython.parallel.apps.ipcontrollerapp.IPCController attribute), 548
mro (IPython.utils.traits.MetaHasTraits attribute), name (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 549
msg_ids (IPython.parallel.client.asyncresult.AsyncHubResult attribute), 550
msg_ids (IPython.parallel.client.asyncresult.AsyncMapResult attribute), 590
msg_ids (IPython.parallel.client.asyncresult.AsyncResult attribute), 596
multi_line_specials (IPython.core.prefilter.PrefilterManager attribute), 601
MultiLineMagicChecker (class in IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 607
multiple_replace() (in module IPython.core.prompts), 532
MultipleInstanceError (class in IPython.parallel.apps.ipcontrollerapp.IPClusterStop attribute), 612
mute_warn() (in module IPython.testing.tools), 848
mutex_opts() (in module IPython.utils.attic), 851
mux (IPython.parallel.controller.hub.HubFactory attribute), 793
name (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 601
name (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 625
name (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 633
name (IPython.parallel.apps.win32support.ForwarderThread attribute), 721
n (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 601
n (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 607
native_line_ends() (in module IPython.utils.text), 826

N

n (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 601
Namespace (class in IPython.parallel.util), 826
native_line_ends() (in module IPython.utils.text), 826

| | | | |
|---|-------------------------------------|---|--|
| | 898 | | |
| needs_local_scope() | (in module IPython.core.magic), 444 | Notifier (class in IPython.utils.growl), 863 | |
| new() (IPython.lib.backgroundjobs.BackgroundJobManager method), 551 | | notifier (IPython.parallel.controller.hub.Hub attribute), 789 | |
| new_do_down() (IPython.core.debugger.Pdb method), 312 | | notifier_port (IPython.parallel.controller.hub.HubFactory attribute), 793 | |
| new_do_frame() (IPython.core.debugger.Pdb method), 312 | | notifier_stream (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798 | |
| new_do_quit() (IPython.core.debugger.Pdb method), 312 | | notify() (in module IPython.utils.growl), 864 | |
| new_do_restart() (IPython.core.debugger.Pdb method), 312 | | notify() (IPython.utils.growl.Notifier method), 863 | |
| new_do_up() (IPython.core.debugger.Pdb method), 312 | | notify_deferred() (in module IPython.utils.growl), 864 | |
| new_main_mod() (IPython.core.interactiveshell.InteractiveShell method), 407 | | notify_deferred() (IPython.utils.growl.Notifier method), 863 | |
| new_session() (IPython.core.history.HistoryManager method), 361 | | notify_start() (IPython.parallel.apps.launcher.BaseLauncherInteractiveShellmethod), 637 | |
| newline (IPython.core.formatters.PlainTextFormatter attribute), 353 | | notify_start() (IPython.parallel.apps.launcher.BatchSystemLauncher method), 640 | |
| NLprinter (class in IPython.utils.io), 865 | | notify_start() (IPython.parallel.apps.launcher.IPClusterLauncher method), 644 | |
| nlstr (IPython.utils.text.LSString attribute), 893 | | notify_start() (IPython.parallel.apps.launcher.LocalControllerLaunch method), 657 | |
| nlstr (IPython.utils.text.SList attribute), 896 | | notify_start() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 660 | |
| no_op() (in module IPython.core.interactiveshell), 417 | | notify_start() (IPython.parallel.apps.launcher.LocalEngineSetLaunch method), 663 | |
| NoColor (IPython.utils.coloransi.InputTermColors attribute), 857 | | notify_start() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 665 | |
| NoColor (IPython.utils.coloransi.TermColors attribute), 858 | | notify_start() (IPython.parallel.apps.launcher.LSFControllerLaunch method), 647 | |
| NoDefaultSpecified (in module IPython.utils.traits), 925 | | notify_start() (IPython.parallel.apps.launcher.LSFEngineSetLaunch method), 650 | |
| NoEnginesRegistered (class in IPython.parallel.error), 818 | | notify_start() (IPython.parallel.apps.launcher.LSFLauncher method), 654 | |
| nohelp (IPython.core.debugger.Pdb attribute), 312 | | notify_start() (IPython.parallel.apps.launcher.MPIExecControllerLaunch method), 668 | |
| noinfo() (IPython.core.oinspect.Inspector method), 449 | | notify_start() (IPython.parallel.apps.launcher.MPIExecEngineSetLaunch method), 671 | |
| Normal (IPython.utils.coloransi.InputTermColors attribute), 857 | | notify_start() (IPython.parallel.apps.launcher.MPIExecLauncher method), 674 | |
| Normal (IPython.utils.coloransi.TermColors attribute), 858 | | notify_start() (IPython.parallel.apps.launcher.PBSControllerLaunch method), 677 | |
| NotAPendingResult (class in IPython.parallel.error), 818 | | notify_start() (IPython.parallel.apps.launcher.PBSEngineSetLaunch method), 681 | |
| NotDefined (class in IPython.parallel.error), 819 | | notify_start() (IPython.parallel.apps.launcher.PBSLauncher method), 684 | |
| NotGiven (class in IPython.utils.attic), 850 | | notify_start() (IPython.parallel.apps.launcher.SGEControllerLaunch method), 688 | |
| NotificationCenter (class in IPython.utils.notification), 875 | | | |
| NotificationError (class in IPython.utils.notification), 875 | | | |

notify_start() (IPython.parallel.apps.launcher.SGEEngineSetLauncher),
 method), 691
notify_start() (IPython.parallel.apps.launcher.SGELauncher),
 method), 695
notify_start() (IPython.parallel.apps.launcher.SSHController),
 method), 698
notify_start() (IPython.parallel.apps.launcher.SSHEngineSetLauncher),
 method), 692
notify_start() (IPython.parallel.apps.launcher.SSHEngine),
 method), 695
notify_start() (IPython.parallel.apps.launcher.SSHEngineSetLauncher),
 method), 698
notify_start() (IPython.parallel.apps.launcher.SSHLauncher),
 method), 701
notify_start() (IPython.parallel.apps.launcher.WindowsHPCCController),
 method), 704
notify_start() (IPython.parallel.apps.launcher.WindowsHPCEngineSet),
 method), 704
notify_start() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 707
notify_start() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 710
notify_start() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 713
notify_start() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 716
notify_stop() (IPython.parallel.apps.launcher.BaseLauncher),
 method), 637
notify_stop() (IPython.parallel.apps.launcher.BatchSystem),
 method), 641
notify_stop() (IPython.parallel.apps.launcher.IPClusterManager),
 method), 644
notify_stop() (IPython.parallel.apps.launcher.LocalController),
 method), 657
notify_stop() (IPython.parallel.apps.launcher.LocalEngineLauncher),
 method), 660
notify_stop() (IPython.parallel.apps.launcher.LocalEnginesetInfo),
 method), 663
notify_stop() (IPython.parallel.apps.launcher.LocalProcessLauncher),
 method), 665
notify_stop() (IPython.parallel.apps.launcher.LSFController),
 method), 647
notify_stop() (IPython.parallel.apps.launcher.LSFEngine),
 method), 650
notify_stop() (IPython.parallel.apps.launcher.LSFLauncher),
 method), 654
notify_stop() (IPython.parallel.apps.launcher.MPIExecController),
 method), 668
notify_stop() (IPython.parallel.apps.launcher.MPIExecEngineSet),
 method), 671
notify_stop() (IPython.parallel.apps.launcher.MPIExecLauncher),
 method), 674
notify_stop() (IPython.parallel.apps.launcher.PBSCController),
 method), 677
notify_stop() (IPython.parallel.apps.launcher.PBSEngineSetLauncher),
 method), 681
notify_stop() (IPython.parallel.apps.launcher.Python.core.inputsplitter),
 method), 373
notify_stop() (IPython.parallel.apps.launcher.SSHLauncher),
 method), 704
notify_stop() (IPython.parallel.apps.launcher.WindowsHPCCController),
 method), 710
notify_stop() (IPython.parallel.apps.launcher.WindowsHPCEngineSet),
 method), 713
notify_stop() (IPython.parallel.apps.launcher.WindowsHPCLauncher),
 method), 716
num_ini_spaces() (in module IPython.utils.text), 898
object_info_string_level (in module IPython.core.oinspect), 452
object_info_string_level (IPython.core.interactiveshell.InteractiveShell),
 attribute), 408
object_info_string_level (IPython.core.interactiveshell.InteractiveShell),
 method), 408
ObjectNumber (class in IPython.utils.traits), 925
ofind() (IPython.core.prefilter.LineInfo method), 488
on_error() (IPython.core.magic), 444
on_probation (IPython.parallel.controller.heartmonitor.HeartMonitor),
 method), 783
on_stop() (IPython.parallel.apps.launcher.BaseLauncher),
 method), 637
on_stop() (IPython.parallel.apps.launcher.BatchSystemLauncher),
 method), 641
on_stop() (IPython.parallel.apps.launcher.IPClusterLauncher),
 method), 644
on_stop() (IPython.parallel.apps.launcher.LocalController),
 method), 657
on_stop() (IPython.parallel.apps.launcher.LocalControllerLauncher),
 method), 657

on_stop() (IPython.parallel.apps.launcher.LocalEngine) [bautechange\(\)](#) (IPython.config.configurable.LoggingConfigurable method), 660
method), 272

on_stop() (IPython.parallel.apps.launcher.LocalEngine) [SetTraitChange\(\)](#) (IPython.config.configurable.SingletonConfigurable method), 663
method), 274

on_stop() (IPython.parallel.apps.launcher.LocalProcess) [dmutechange\(\)](#) (IPython.core.alias.AliasManager method), 665
method), 285

on_stop() (IPython.parallel.apps.launcher.LSFController) [enLanuchchange\(\)](#) (IPython.core.application.BaseIPythonApplication method), 647
method), 290

on_stop() (IPython.parallel.apps.launcher.LSFEngine) [SetLanuchchange\(\)](#) (IPython.core.builtin_trap.BuiltinTrap method), 650
method), 295

on_stop() (IPython.parallel.apps.launcher.LSFLauncher) [on_trait_change\(\)](#) (IPython.core.display_trap.DisplayTrap method), 654
method), 317

on_stop() (IPython.parallel.apps.launcher.MPIExecController) [TraitChange\(\)](#) (IPython.core.displayhook.DisplayHook method), 668
method), 320

on_stop() (IPython.parallel.apps.launcher.MPIExecEngine) [SetTraitChange\(\)](#) (IPython.core.displaypub.DisplayPublisher method), 671
method), 323

on_stop() (IPython.parallel.apps.launcher.MPIExecLauncher) [on_trait_change\(\)](#) (IPython.core.extensions.ExtensionManager method), 674
method), 329

on_stop() (IPython.parallel.apps.launcher.PBSController) [enLanuchchange\(\)](#) (IPython.core.formatters.BaseFormatter method), 678
method), 333

on_stop() (IPython.parallel.apps.launcher.PBSEngine) [SetLanuchchange\(\)](#) (IPython.core.formatters.DisplayFormatter method), 681
method), 336

on_stop() (IPython.parallel.apps.launcher.PBSLauncher) [on_trait_change\(\)](#) (IPython.core.formatters.HTMLFormatter method), 684
method), 339

on_stop() (IPython.parallel.apps.launcher.SGEController) [enLanuchchange\(\)](#) (IPython.core.formatters.JavascriptFormatter method), 688
method), 344

on_stop() (IPython.parallel.apps.launcher.SGEEngine) [SetLanuchchange\(\)](#) (IPython.core.formatters.JSONFormatter method), 692
method), 341

on_stop() (IPython.parallel.apps.launcher.SGELauncher) [on_trait_change\(\)](#) (IPython.core.formatters.LatexFormatter method), 695
method), 347

on_stop() (IPython.parallel.apps.launcher.SSHController) [TraitChange\(\)](#) (IPython.core.formatters.PlainTextFormatter method), 698
method), 353

on_stop() (IPython.parallel.apps.launcher.SSHEngine) [Lanuchtrait_change\(\)](#) (IPython.core.formatters.PNGFormatter method), 701
method), 350

on_stop() (IPython.parallel.apps.launcher.SSHEngine) [SetTraitChange\(\)](#) (IPython.core.formatters.SVGFormatter method), 704
method), 356

on_stop() (IPython.parallel.apps.launcher.SSHLauncher) [on_trait_change\(\)](#) (IPython.core.history.HistoryManager method), 707
method), 361

on_stop() (IPython.parallel.apps.launcher.WindowsHP6Controller) [TraitChange\(\)](#) (IPython.core.interactiveshell.InteractiveShell method), 710
method), 408

on_stop() (IPython.parallel.apps.launcher.WindowsHP6Engine) [SetTraitChange\(\)](#) (IPython.core.payload.PayloadManager method), 713
method), 455

on_stop() (IPython.parallel.apps.launcher.WindowsHP6Launcher) [TraitChange\(\)](#) (IPython.core.plugin.Plugin method), 716
method), 458

on_trait_change() (IPython.config.application.Application) [on_trait_change\(\)](#) (IPython.core.plugin.PluginManager method), 265
method), 459

on_trait_change() (IPython.config.configurable.Configurable) [TraitChange\(\)](#) (IPython.core.prefilter.AliasChecker method), 269
method), 462

on_trait_change() (IPython.core.prefilter.AliasHandler on_trait_change() (IPython.core.prefilter.ShellEscapeHandler method), 464
method), 511

on_trait_change() (IPython.core.prefilter.AssignMagicTransformer on_trait_change() (IPython.core.profileapp.ProfileApp method), 466
method), 515

on_trait_change() (IPython.core.prefilter.AssignmentCheckTraitChange on_trait_change() (IPython.core.profileapp.ProfileCreate method), 470
method), 519

on_trait_change() (IPython.core.prefilter.AssignSystemTransformer on_trait_change() (IPython.core.profileapp.ProfileList method), 468
method), 523

on_trait_change() (IPython.core.prefilter.AutocallChecker on_trait_change() (IPython.core.profiledir.ProfileDir method), 475
method), 528

on_trait_change() (IPython.core.prefilter.AutoHandler on_trait_change() (IPython.core.shellapp.InteractiveShellApp method), 472
method), 534

on_trait_change() (IPython.core.prefilter.AutoMagicCheckTraitChange on_trait_change() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 473
method), 590

on_trait_change() (IPython.core.prefilter.EmacsCheckTraitChange on_trait_change() (IPython.parallel.apps.ipclusterapp.IPClusterApp method), 477
method), 596

on_trait_change() (IPython.core.prefilter.EmacsHandle on_trait_change() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 479
method), 601

on_trait_change() (IPython.core.prefilter.EscCharsCheckTraitChange on_trait_change() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 481
method), 607

on_trait_change() (IPython.core.prefilter.HelpHandler on_trait_change() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 483
method), 612

on_trait_change() (IPython.core.prefilter.IPyAutocallCheckTraitChange on_trait_change() (IPython.parallel.apps.ipcontrollerapp.IPController method), 484
method), 618

on_trait_change() (IPython.core.prefilter.IPyPromptTransformer on_trait_change() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 486
method), 625

on_trait_change() (IPython.core.prefilter.MacroCheckTraitChange on_trait_change() (IPython.parallel.apps.ipengineapp.MPI method), 489
method), 629

on_trait_change() (IPython.core.prefilter.MacroHandle on_trait_change() (IPython.parallel.apps.iploggerapp.IPLoggerApp method), 491
method), 633

on_trait_change() (IPython.core.prefilter.MagicHandle on_trait_change() (IPython.parallel.apps.launcher.BaseLauncher method), 492
method), 637

on_trait_change() (IPython.core.prefilter.MultiLineMagicTraitChange on_trait_change() (IPython.parallel.apps.launcher.BatchSystemLauncher method), 494
method), 641

on_trait_change() (IPython.core.prefilter.PrefilterChecker on_trait_change() (IPython.parallel.apps.launcher.IPClusterLauncher method), 496
method), 644

on_trait_change() (IPython.core.prefilter.PrefilterHandle on_trait_change() (IPython.parallel.apps.launcher.LocalControllerLauncher method), 498
method), 657

on_trait_change() (IPython.core.prefilter.PrefilterManager on_trait_change() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 501
method), 660

on_trait_change() (IPython.core.prefilter.PrefilterTransformer on_trait_change() (IPython.parallel.apps.launcher.LocalEngineSetLauncher method), 503
method), 663

on_trait_change() (IPython.core.prefilter.PyPromptTransformer on_trait_change() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 505
method), 666

on_trait_change() (IPython.core.prefilter.PythonOpsCheckTraitChange on_trait_change() (IPython.parallel.apps.launcher.LSFControllerLauncher method), 507
method), 647

on_trait_change() (IPython.core.prefilter.ShellEscapeCheckTraitChange on_trait_change() (IPython.parallel.apps.launcher.LSFEngineSetLauncher method), 509
method), 651

on_trait_change() (IPython.parallel.apps.launcher.LSFLauncher),
 method), 654

on_trait_change() (IPython.parallel.apps.launcher.MPIEngineController),
 method), 669

on_trait_change() (IPython.parallel.apps.launcher.MPIEngineSet),
 method), 671

on_trait_change() (IPython.parallel.apps.launcher.MPIEngineTask),
 method), 674

on_trait_change() (IPython.parallel.apps.launcher.PBSController),
 method), 678

on_trait_change() (IPython.parallel.apps.launcher.PBSEngineSet),
 method), 681

on_trait_change() (IPython.parallel.apps.launcher.PBSHeartMonitor),
 method), 685

on_trait_change() (IPython.parallel.apps.launcher.SGEController),
 method), 688

on_trait_change() (IPython.parallel.apps.launcher.SGEHub),
 method), 692

on_trait_change() (IPython.parallel.apps.launcher.SGEHubFactory),
 method), 695

on_trait_change() (IPython.parallel.apps.launcher.SSHController),
 method), 698

on_trait_change() (IPython.parallel.apps.launcher.SSHSQLiteHandler),
 method), 701

on_trait_change() (IPython.parallel.apps.launcher.SSHEngineSet),
 method), 704

on_trait_change() (IPython.parallel.apps.launcher.SSHKernel),
 method), 707

on_trait_change() (IPython.parallel.apps.launcher.WindowsHPCChange),
 method), 710

on_trait_change() (IPython.parallel.apps.launcher.WindowsHPCEngineSet),
 method), 713

on_trait_change() (IPython.parallel.apps.launcher.WindowsHPCOneTimeProperty),
 method), 716

on_trait_change() (IPython.parallel.apps.logwatcher.LogWatcher),
 method), 719

on_trait_change() (IPython.parallel.apps.winhpcjob.IPOptionDescription),
 method), 723

on_trait_change() (IPython.parallel.apps.winhpcjob.IPOptionTaskDescription),
 method), 726

on_trait_change() (IPython.parallel.apps.winhpcjob.IPOptionSetDescription),
 method), 729

on_trait_change() (IPython.parallel.apps.winhpcjob.IPOptionTaskDescription),
 method), 732

on_trait_change() (IPython.parallel.apps.winhpcjob.WinHPCJobDescription),
 method), 735

on_trait_change() (IPython.parallel.apps.winhpcjob.WinHPCTaskDescription),
 method), 738

on_trait_change() (IPython.parallel.client.Client),
 method), 748

on_trait_change() (IPython.parallel.client.view.DirectView),
 method), 759

on_trait_change() (IPython.parallel.client.view.LoadBalancedView),
 method), 765

on_trait_change() (IPython.parallel.client.view.View),
 method), 769

on_trait_change() (IPython.parallel.controller.dictdb.BaseDB),
 method), 777

on_trait_change() (IPython.parallel.controller.dictdb.DictDB),
 method), 780

on_trait_change() (IPython.parallel.controller.heartmonitor.HeartMonitor),
 method), 783

on_trait_change() (IPython.parallel.controller.hub.EngineConnector),
 method), 785

on_trait_change() (IPython.parallel.controller.hub.Hub),
 method), 789

on_trait_change() (IPython.parallel.controller.hub.HubFactory),
 method), 793

on_trait_change() (IPython.parallel.controller.scheduler.TaskScheduler),
 method), 798

on_trait_change() (IPython.parallel.controller.sqlite.SQLiteDB),
 method), 802

on_trait_change() (IPython.parallel.engine.EngineFactory),
 method), 804

on_trait_change() (IPython.parallel.engine.streamkernel.Kernel),
 method), 810

on_trait_change() (IPython.parallel.factory.RegistrationFactory),
 method), 824

on_trait_change() (IPython.utils.traits.HasTraits),
 method), 919

on_trait_change() (IPython.core.debugger.Pdb),
 method), 312

 OneTimeProperty (class in IPython.utils.autoattr), 312

onlyif() (in module IPython.testing.decorators), 831

on_trait_change() (IPython.config.application.Application),
 attribute), 266

on_trait_change() (IPython.core.application.BaseIPythonApplication),
 attribute), 290

on_trait_change() (IPython.core.profileapp.ProfileApp),
 attribute), 515

on_trait_change() (IPython.core.profileapp.ProfileCreate),
 attribute), 520

on_trait_change() (IPython.core.profileapp.ProfileList),
 attribute), 524

on_trait_change() (IPython.parallel.apps.baseapp.BaseParallelApplication),
 attribute), 591

option_description (IPython.parallel.apps.ipclusterapp.IPClusterAttribute), 291
 attribute), 596
option_description (IPython.parallel.apps.ipclusterapp.IPClusterEngine), 520
 attribute), 601
option_description (IPython.parallel.apps.ipclusterapp.IPClusterAttribute), 591
 attribute), 607
option_description (IPython.parallel.apps.ipclusterapp.IPClusterStopAttribute), 601
 attribute), 613
option_description (IPython.parallel.apps.ipcontrollerapp.IPControllerApp), 607
 attribute), 619
option_description (IPython.parallel.apps.ipengineapp.IPEngineAttribute), 613
 attribute), 625
option_description (IPython.parallel.apps.iploggerapp.IPLLoggerAttribute), 619
 attribute), 633
ostream (IPython.core.ultratb.AutoFormattedTB attribute), 538
ostream (IPython.core.ultratb.ColorTB attribute), 539
ostream (IPython.core.ultratb.FormattedTB attribute), 541
ostream (IPython.core.ultratb.ListTB attribute), 542
ostream (IPython.core.ultratb.SyntaxTB attribute), 544
ostream (IPython.core.ultratb.TBTools attribute), 545
ostream (IPython.core.ultratb.VerboseTB attribute), 546
osx_clipboard_get() (in module IPython.lib.clipboard), 553
out_stream_factory (IPython.parallel.engine.engine.EngineFactory), 805
output() (IPython.lib.pretty.Breakable method), 582
output() (IPython.lib.pretty.Group method), 582
output() (IPython.lib.pretty.Printable method), 583
output() (IPython.lib.pretty.Text method), 584
output_hist (IPython.core.history.HistoryManager attribute), 362
output_hist_reprs (IPython.core.history.HistoryManager attribute), 362
outstanding (IPython.parallel.client.client.Client attribute), 749
outstanding (IPython.parallel.client.view.DirectView attribute), 760
outstanding (IPython.parallel.client.view.LoadBalancedView attribute), 765
outstanding (IPython.parallel.client.view.View attribute), 769
overwrite (IPython.core.application.BaseIPythonApplication attribute), 291
 overwrite (IPython.core.profileapp.ProfileCreate attribute), 520
 overwrite (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 591
 overwrite (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 601
 overwrite (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 607
 overwrite (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 613
 overwrite (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 619
 overwrite (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 625
 overwrite (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 634
owner (IPython.parallel.apps.winhpcjob.IPCControllerJob attribute), 724
owner (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 729
owner (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 735

P

p (IPython.utils.text.LSString attribute), 893
p (IPython.utils.text.SList attribute), 896
p_template (IPython.core.prompts.BasePrompt attribute), 530
p_template (IPython.core.prompts.Prompt1 attribute), 531
p_template (IPython.core.prompts.Prompt2 attribute), 531
p_template (IPython.core.prompts.PromptOut attribute), 532
pack_apply_message() (in module IPython.parallel.util), 829
page() (in module IPython.core.page), 452
page() (in module IPython.core.payloadpage), 456
page_dumb() (in module IPython.core.page), 453
page_file() (in module IPython.core.page), 453
parallel (IPython.core.profileapp.ProfileCreate attribute), 520
parallel() (in module IPython.parallel.client.remotefunction), 756
parallel() (IPython.parallel.client.view.DirectView method), 760

```

parallel() (IPython.parallel.client.view.LoadBalancedView)
    pause_command_line()
        (IPython.parallel.apps.ipclusterapp.IPClusterStop
            method), 765
parallel() (IPython.parallel.client.view.View)
    pause_command_line()
        (IPython.parallel.apps.ipclusterapp.IPClusterStop
            method), 613
    method), 770
ParallelErrorHandler (class) in parse_command_line()
    (IPython.parallel.apps.baseapp), 593
    (IPython.parallel.apps.ipcontrollerapp.IPClusterApp
        method), 619
ParallelFunction (class) in parse_command_line()
    (IPython.parallel.client.remotefunction),
        (IPython.parallel.apps.ipengineapp.IPEngineApp
            method), 625
    754
    754
params (IPython.testing.iptest.IPTester attribute), parse_command_line()
    835
    835
parse() (IPython.utils.text.EvalFormatter method), parse_command_line()
    890
    890
parse_args() (IPython.config.loader.ArgumentParser)
    method), 278
    278
parse_args() (IPython.core.magic_arguments.MagicArgumentParser)
    method), 446
    446
parse_argstring() (in module IPython.core.magic_arguments), 448
parse_argstring() (IPython.core.magic_arguments.MagicArgumentParser)
    method), 446
    446
parse_command_line()
    (IPython.config.application.Application
        method), 266
    266
parse_command_line()
    (IPython.core.application.BaseIPythonApplication
        method), 291
    291
parse_command_line()
    (IPython.core.profileapp.ProfileApp
        method), 515
    515
parse_command_line()
    (IPython.core.profileapp.ProfileCreate
        method), 520
    520
parse_command_line()
    (IPython.core.profileapp.ProfileList
        method), 524
    524
parse_command_line()
    (IPython.parallel.apps.baseapp.BaseParallelApplication
        method), 591
    591
parse_command_line()
    (IPython.parallel.apps.ipclusterapp.IPClusterApp
        method), 596
    596
parse_command_line()
    (IPython.parallel.apps.ipclusterapp.IPClusterEngines
        method), 446
    446
    601
parse_command_line()
    (IPython.parallel.apps.ipclusterapp.IPClusterStart
        method), 607
    607
    start_options() (IPython.core.interactiveshell.InteractiveShell
        method), 408
    408
    parse_known_args()
        (IPython.config.loader.ArgumentParser
            method), 278
    278
    parse_known_args()
        (IPython.core.magic_arguments.MagicArgumentParser
            method), 446
    446
    parse_notifier_name()
        (in module IPython.utils.traitlets), 934
    934
    934

```

parse_options() (IPython.core.magic.Magic method), 443
parse_test_output() (in module IPython.testing.tools), 848
parseline() (IPython.core.debugger.Pdb method), 312
Parser (class in IPython.utils.PyColorize), 849
partition (IPython.utils.text.LSString attribute), 893
PATH, 2
paths (IPython.utils.text.LSString attribute), 893
paths (IPython.utils.text.SList attribute), 896
payload_manager (IPython.core.interactiveshell.InteractiveShell attribute), 408
PayloadManager (class in IPython.core.payload), 454
PBMessageSizeError (class in IPython.parallel.error), 819
PBSControllerLauncher (class in IPython.parallel.apps.launcher), 676
PBSEngineSetLauncher (class in IPython.parallel.apps.launcher), 679
PBSLauncher (class in IPython.parallel.apps.launcher), 683
Pdb (class in IPython.core.debugger), 306
pdb (IPython.core.interactiveshell.InteractiveShell attribute), 408
pdef() (IPython.core.oinspect.Inspector method), 449
pdoc() (IPython.core.oinspect.Inspector method), 450
pending (IPython.parallel.controller.hub.EngineConnector attribute), 785
pending (IPython.parallel.controller.hub.Hub attribute), 789
pending (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798
period (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 783
pexpect_monkeypatch() (in module IPython.lib.irunner), 578
pfile() (IPython.core.oinspect.Inspector method), 450
PickleShareDB (class in IPython.utils.pickleshare), 880
PickleShareLink (class in IPython.utils.pickleshare), 882
pid_dir (IPython.core.profiledir.ProfileDir attribute), 528
pid_dir_name (IPython.core.profiledir.ProfileDir attribute), 528
PIDFileError (class in IPython.parallel.apps.baseapp), 593
pids (IPython.testing.iptest.IPTester attribute), 835
pinfo() (IPython.core.oinspect.Inspector method), 450
pinfo_fields1 (IPython.core.oinspect.Inspector attribute), 450
pinfo_fields_obj (IPython.core.oinspect.Inspector attribute), 450
pingShell (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 783
pkg_commit_hash() (in module IPython.utils.sysinfo), 886
pkg_info() (in module IPython.utils.sysinfo), 887
plain() (IPython.core.ultratb.AutoFormattedTB method), 538
plain() (IPython.core.ultratb.ColorTB method), 540
plain() (IPython.core.ultratb.FormattedTB method), 541
plain_text_only (IPython.core.formatters.DisplayFormatter attribute), 336
plainrandom() (in module IPython.parallel.controller.scheduler), 799
PlainTextFormatter (class in IPython.core.formatters), 351
Plugin (class in IPython.core.plugin), 457
plugin_manager (IPython.core.interactiveshell.InteractiveShell attribute), 408
PluginManager (class in IPython.core.plugin), 459
plugins (IPython.core.plugin.PluginManager attribute), 460
PNGFormatter (class in IPython.core.formatters), 348
poll() (IPython.parallel.apps.launcher.IPClusterLauncher method), 644
poll() (IPython.parallel.apps.launcher.LocalControllerLauncher method), 658
poll() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 661
poll() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 666
poll() (IPython.parallel.apps.launcher.MPIExecControllerLauncher method), 669
poll() (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher method), 672

poll() (IPython.parallel.apps.launcher.MPIExecLauncher method), 675
popitem() (IPython.parallel.client.client.Metadata attribute), 752
poll() (IPython.parallel.apps.launcher.SSHControllerLauncher method), 699
popitem() (IPython.parallel.util.Namespace attribute), 827
poll() (IPython.parallel.apps.launcher.SSHEngineLauncher method), 702
popitem() (IPython.parallel.util.ReverseDict attribute), 828
poll() (IPython.parallel.apps.launcher.SSHLauncher method), 707
popitem() (IPython.testing.globalipapp.ipnsdict attribute), 834
poll_frequency() (IPython.parallel.apps.launcher.IPCluster attribute), 644
popitem() (IPython.utils.coloransi.ColorSchemeTable attribute), 856
poll_frequency() (IPython.parallel.apps.launcher.LocalCluster attribute), 871
popitem() (IPython.utils.pickleshare.PickleShareDB attribute), 871
poll_frequency() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 881
popkey() (in module IPython.utils.attic), 851
poll_frequency() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 558
poll_frequency() (IPython.parallel.apps.launcher.MPIExecController method), 560
poll_frequency() (IPython.parallel.apps.launcher.MPIExecEngineLauncher method), 562
post_cmd() (IPython.lib.demo.IPythonDemo method), 562
poll_frequency() (IPython.parallel.apps.launcher.MPIExecLauncher method), 564
post_cmd() (IPython.lib.demo.IPythonLineDemo method), 564
poll_frequency() (IPython.parallel.apps.launcher.SSHControllerLauncher method), 565
post_cmd() (IPython.lib.demo.LineDemo method), 565
poll_frequency() (IPython.parallel.apps.launcher.SSHEngineLauncher method), 565
post_notification() (IPython.utils.notification.NotificationCenter method), 565
poll_frequency() (IPython.parallel.apps.launcher.SSHLauncher method), 876
postcmd() (IPython.core.debugger.Pdb method), 312
pongstream() (IPython.parallel.controller.heartmonitor.HeartMonitor method), 312
pprint() (IPython.core.formatters.PlainTextFormatter attribute), 354
pop() (IPython.config.loader.Config attribute), 279
pop() (IPython.parallel.client.client.Metadata attribute), 752
pop() (IPython.parallel.controller.dependency.Dependency attribute), 774
pop() (IPython.parallel.util.Namespace attribute), 827
pop() (IPython.testing.globalipapp.ipnsdict attribute), 834
pop() (IPython.utils.coloransi.ColorSchemeTable attribute), 856
pop() (IPython.utils.ipstruct.Struct attribute), 871
pop() (IPython.utils.text.SList attribute), 896
pop() (IPython.parallel.util.ReverseDict method), 828
pop() (IPython.utils.pickleshare.PickleShareDB method), 881
popitem() (IPython.config.loader.Config attribute), 279
popitem() (IPython.parallel.client.client.Metadata attribute), 752
popitem() (IPython.parallel.util.Namespace attribute), 827
popitem() (IPython.parallel.util.ReverseDict attribute), 828
popitem() (IPython.utils.pickleshare.PickleShareDB attribute), 881
pre_prompt_hook() (in module IPython.core.hooks), 367
pre_readline() (IPython.core.interactiveshell.InteractiveShell method), 567

method), 409
pre_run_code_hook() (in module IPython.core.hooks), 368
precmd() (IPython.core.debugger.Pdb method), 312
prefilter_line() (IPython.core.prefilter.PrefilterManager.prefilter_manager (IPython.core.prefilter.PrefilterTransformer method), 501
prefilter_line_info() (IPython.core.prefilter.PrefilterManager.prefilter_manager (IPython.core.prefilter.PyPromptTransformer method), 501
prefilter_lines() (IPython.core.prefilter.PrefilterManager.prefilter_manager (IPython.core.prefilter.PythonOpsChecker method), 501
prefilter_manager (IPython.core.interactiveshell.InteractiveShellManager (IPython.core.prefilter.ShellEscapeChecker attribute), 409
prefilter_manager (IPython.core.prefilter.AliasChecker.prefilter_manager (IPython.core.prefilter.ShellEscapeHandler attribute), 463
prefilter_manager (IPython.core.prefilter.AliasHandler PrefilterChecker (class in IPython.core.prefilter), 495
prefilter_manager (IPython.core.prefilter.AssignMagicPrefilterError (class in IPython.core.prefilter), 497
attribute), 466
prefilter_manager (IPython.core.prefilter.AssignmentPrefilterManager (class in IPython.core.prefilter), 499
attribute), 470
prefilter_manager (IPython.core.prefilter.AssignSystemPrefilterTransformer (class in IPython.core.prefilter), 503
attribute), 468
prefilter_manager (IPython.core.prefilter.AutocallCheckerPrefilterLoop (IPython.core.debugger.Pdb method), 312
attribute), 476
prefilter_manager (IPython.core.prefilter.AutoHandler (IPython.utils.syspathcontext), 888
attribute), 472
pretty() (in module IPython.lib.pretty), 585
prefilter_manager (IPython.core.prefilter.AutoMagicChecker) (IPython.lib.pretty.RepresentationPrinter
attribute), 474
PrettyPrinter (class in IPython.lib.pretty), 582
attribute), 477
print_alias_help() (IPython.config.application.Application
prefilter_manager (IPython.core.prefilter.EmacsHandler method), 266
attribute), 479
print_alias_help() (IPython.core.application.BaseIPythonApplication
prefilter_manager (IPython.core.prefilter.EscCharsChecker method), 291
attribute), 481
print_alias_help() (IPython.core.profileapp.ProfileApp
prefilter_manager (IPython.core.prefilter.HelpHandler method), 515
attribute), 483
print_alias_help() (IPython.core.profileapp.ProfileCreate
prefilter_manager (IPython.core.prefilter.IPyAutocallChecker method), 520
attribute), 485
print_alias_help() (IPython.core.profileapp.ProfileList
prefilter_manager (IPython.core.prefilter.IPyPromptTransformer method), 524
attribute), 487
print_alias_help() (IPython.parallel.apps.baseapp.BaseParallelApp
prefilter_manager (IPython.core.prefilter.MacroChecker method), 591
attribute), 489
print_alias_help() (IPython.parallel.apps.ipclusterapp.IPClusterApp
prefilter_manager (IPython.core.prefilter.MacroHandler method), 596
attribute), 491
print_alias_help() (IPython.parallel.apps.ipclusterapp.IPClusterEngine
prefilter_manager (IPython.core.prefilter.MagicHandler method), 601
attribute), 493
print_alias_help() (IPython.parallel.apps.ipclusterapp.IPClusterStart
prefilter_manager (IPython.core.prefilter.MultiLineMagicChecker) (IPython.parallel.apps.ipclusterapp.IPClusterStop
attribute), 495
print_alias_help() (IPython.parallel.apps.ipclusterapp.IPClusterStop

method), 613
print_alias_help() (IPython.parallel.apps.ipcontrollerapp.IPClusterStart method), 619
print_alias_help() (IPython.parallel.apps.ipengineapp.IPEngineApp) (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 625
print_alias_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp) (IPython.parallel.apps.ipcontrollerapp.IPController method), 634
print_description() (IPython.config.application.Application) print_examples() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 266
print_description() (IPython.core.application.BaseIPythonApplication) print_examples() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 291
print_description() (IPython.core.profileapp.ProfileApp) print_figure() (in module IPython.lib.pylabtools), 586
print_description() (IPython.core.profileapp.ProfileCreate) print_flag_help() (IPython.config.application.Application method), 520
print_description() (IPython.core.profileapp.ProfileList) print_flag_help() (IPython.core.application.BaseIPythonApplication method), 524
print_description() (IPython.parallel.apps.baseapp.BaseParallelApp) print_flag_help() (IPython.core.profileapp.ProfileApp method), 591
print_description() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_flag_help() (IPython.core.profileapp.ProfileCreate method), 596
print_description() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_flag_help() (IPython.core.profileapp.ProfileList method), 601
print_description() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_flag_help() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 607
print_description() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_flag_help() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 613
print_description() (IPython.parallel.apps.iploggerapp.IPLLoggerApp) print_flag_help() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 634
print_examples() (IPython.config.application.Application) print_flag_help() (IPython.parallel.apps.ipcontrollerapp.IPController method), 266
print_examples() (IPython.core.application.BaseIPythonApplication) print_flag_help() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 291
print_examples() (IPython.core.profileapp.ProfileApp) print_flag_help() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 515
print_examples() (IPython.core.profileapp.ProfileCreate) print_flag_help() (IPython.config.application.Application method), 520
print_examples() (IPython.core.profileapp.ProfileList) print_flag_help() (IPython.config.loader.ArgumentParser method), 524
print_examples() (IPython.parallel.apps.baseapp.BaseParallelApp) print_flag_help() (IPython.core.application.BaseIPythonApplication method), 591
print_examples() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_flag_help() (IPython.core.magic_arguments.MagicArgumentParser method), 596
print_examples() (IPython.parallel.apps.ipclusterapp.IPClusterApp) print_engines (IPython.core.profileapp.ProfileApp

method), 515
print_help() (IPython.core.profileapp.ProfileCreate method), 520
print_help() (IPython.core.profileapp.ProfileList method), 524
print_help() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 591
print_help() (IPython.parallel.apps.ipclusterapp.IPClusterApp method), 597
print_help() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 602
print_help() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 607
print_help() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 613
print_help() (IPython.parallel.apps.ipcontrollerapp.IPClusterStart method), 619
print_help() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 626
print_help() (IPython.parallel.apps.iploggerapp.IPLoaderApp method), 634
print_help() (IPython.parallel.apps.ipengineapp.IPEngineApp stack_entry) (IPython.core.debugger.Pdb method), 312
print_help() (IPython.parallel.apps.iploggerapp.IPLoaderApp stack_trace) (IPython.core.debugger.Pdb method), 312
print_list_lines() (IPython.core.debugger.Pdb print_subcommands) (IPython.config.application.Application method), 266
print_method (IPython.core.formatters.BaseFormatter attribute), 333
print_method (IPython.core.formatters.HTMLFormatter attribute), 339
print_method (IPython.core.formatters.JavascriptFormatter attribute), 345
print_method (IPython.core.formatters.JSONFormatter attribute), 342
print_method (IPython.core.formatters.LatexFormatter attribute), 348
print_method (IPython.core.formatters.PlainTextFormatter attribute), 354
print_method (IPython.core.formatters.PNGFormatter attribute), 350
print_method (IPython.core.formatters.SVGFormatter attribute), 357
print_options() (IPython.config.application.Application print_subcommands) (IPython.core.profileapp.ProfileCreate method), 266
print_options() (IPython.core.application.BaseIPythonApplication method), 291
print_options() (IPython.core.profileapp.ProfileApp method), 516
print_options() (IPython.core.profileapp.ProfileCreate print_subcommands) (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 520
print_options() (IPython.core.profileapp.ProfileList print_subcommands) (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 607

print_subcommands()
 (IPython.parallel.apps.ipclusterapp.IPClusterStop
 method), 613

print_subcommands()
 (IPython.parallel.apps.ipcontrollerapp.IPControllerApp
 method), 619

print_subcommands()
 (IPython.parallel.apps.ipengineapp.IPEngineApp
 method), 626

print_subcommands()
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 method), 634

print_topics() (IPython.core.debugger.Pdb method), 312

print_tracebacks() (IPython.parallel.error.CompositeError
 method), 813

print_usage() (IPython.config.loader.ArgumentParser
 method), 278

print_usage() (IPython.core.magic_arguments.MagicArgumentParser
 method), 446

print_version() (IPython.config.application.Application
 method), 266

print_version() (IPython.config.loader.ArgumentParser
 method), 278

print_version() (IPython.core.application.BaseIPythonApplication
 method), 291

print_version() (IPython.core.magic_arguments.MagicArgumentParser
 method), 446

print_version() (IPython.core.profileapp.ProfileApp
 method), 516

print_version() (IPython.core.profileapp.ProfileCreate
 method), 520

print_version() (IPython.core.profileapp.ProfileList
 method), 524

print_version() (IPython.parallel.apps.baseapp.BaseParallelApp
 method), 591

print_version() (IPython.parallel.apps.ipclusterapp.IPClusterStop
 method), 597

print_version() (IPython.parallel.apps.ipclusterapp.IPClusterStopEngines
 method), 602

print_version() (IPython.parallel.apps.ipclusterapp.IPClusterStopEngines
 method), 608

print_version() (IPython.parallel.apps.ipclusterapp.IPClusterStopEngines
 method), 613

print_version() (IPython.parallel.apps.ipcontrollerapp.IPClusterStopEngines
 method), 619

print_version() (IPython.parallel.apps.ipengineapp.IPEngineAppState
 method), 626

 print_version() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
 method), 634

 Printable (class in IPython.lib.pretty), 583

 printer() (in module IPython.parallel.engine.streamkernel), 811

 priority (IPython.core.prefilter.AliasChecker
 attribute), 463

 priority (IPython.core.prefilter.AssignMagicTransformer
 attribute), 466

 priority (IPython.core.prefilter.AssignmentChecker
 attribute), 470

 priority (IPython.core.prefilter.AssignSystemTransformer
 attribute), 468

 priority (IPython.core.prefilter.AutocallChecker
 attribute), 476

 priority (IPython.core.prefilter.AutoMagicChecker
 attribute), 474

 priority (IPython.core.prefilter.EmacsChecker
 attribute), 478

 priority (IPython.core.prefilter.EscCharsChecker
 attribute), 481

 priority (IPython.core.prefilter.IPyAutocallChecker
 attribute), 485

 priority (IPython.core.prefilter.IPyPromptTransformer
 attribute), 487

 priority (IPython.core.prefilter.MacroChecker
 attribute), 489

 priority (IPython.core.prefilter.MultiLineMagicChecker
 attribute), 495

 priority (IPython.core.prefilter.PrefilterChecker
 attribute), 496

 priority (IPython.core.prefilter.PrefilterTransformer
 attribute), 504

 priority (IPython.core.prefilter.PyPromptTransformer
 attribute), 506

 priority (IPython.core.prefilter.PythonOpsChecker
 attribute), 508

 priority (IPython.core.prefilter.ShellEscapeChecker
 attribute), 510

 priority (IPython.core.prefilter.StopPython.parallel.apps.winhpcjob.IPControllerJob
 attribute), 724

 priority (IPython.core.prefilter.StopPython.parallel.apps.winhpcjob.IPEngineSetJob
 attribute), 729

 priority (IPython.core.prefilter.WinHPCJob
 attribute), 735

 IPython.parallel.apps.winhpcjob.WinHPCJob
 (class in IPython.parallel.apps.launcher), 686

profile (IPython.core.application.BaseIPythonApplication attribute), 669
profile (IPython.core.interactiveshell.InteractiveShell attribute), 409
profile (IPython.core.profileapp.ProfileCreate attribute), 520
profile (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 699
profile (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 702
profile (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 707
profile (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 613
profile (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 729
profile (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 735
profile (IPython.parallel.apps.iploggerapp.IPLoaderApp attribute), 634
profile (IPython.parallel.client.client.Client attribute), 749
profile_dir (IPython.core.interactiveshell.InteractiveShell attribute), 409
profile_missing_notice()
profile_missing_notice() (IPython.core.magic.Magic method), 444
ProfileApp (class in IPython.core.profileapp), 513
ProfileCreate (class in IPython.core.profileapp), 516
ProfileDir (class in IPython.core.profiledir), 525
ProfileDirError (class in IPython.core.profiledir), 529
ProfileList (class in IPython.core.profileapp), 521
program (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 669
program (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 672
program (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 675
program_args (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 672
program_args (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 675
program_args (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 699
program_args (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 699
program_args (IPython.parallel.apps.launcher.SSHLauncher attribute), 702
program_args (IPython.parallel.apps.launcher.SSHLauncher attribute), 707
project (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 724
project (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 729
project (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 735
prompt (IPython.core.debugger.Pdb attribute), 313
Prompt1 (class in IPython.core.prompts), 530
Prompt2 (class in IPython.core.prompts), 531
prompt_count (IPython.core.displayhook.DisplayHook attribute), 320
prompt_in1 (IPython.core.interactiveshell.InteractiveShell attribute), 409
prompt_in2 (IPython.core.interactiveshell.InteractiveShell attribute), 409
prompt_out (IPython.core.interactiveshell.InteractiveShell attribute), 409
PromptOut (class in IPython.core.prompts), 531
prompts_pad_left (IPython.core.interactiveshell.InteractiveShell attribute), 409
protect_filename() (in module IPython.core.completer), 303
ProtocolError (class in IPython.parallel.error), 819
psearch() (IPython.core.oinspect.Inspector method), 450
psource() (IPython.core.oinspect.Inspector method), 451
publish() (IPython.core.displaypub.DisplayPublisher method), 323
publish_display_data() (in module IPython.core.displaypub), 323
publish_display_pub() (IPython.core.displaypub.DisplayPublisher method), 324
pull() (IPython.parallel.client.view.DirectView method), 760
program (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 699
program (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 760
program (IPython.parallel.apps.launcher.SSHLauncher attribute), 702
program_args (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 670

purge_results() (IPython.parallel.client.view.LoadBalancedView
 method), 765

purge_results() (IPython.parallel.client.view.View
 method), 770

purge_results() (IPython.parallel.controller.hub.Hub
 method), 789

Purple (IPython.utils.coloransi.InputTermColors attribute), 857

Purple (IPython.utils.coloransi.TermColors attribute), 858

push() (IPython.core.inputsplitter.InputSplitter
 method), 371

push() (IPython.core.inputsplitter.IPythonInputSplitter
 method), 370

push() (IPython.core.interactiveshell.InteractiveShell
 method), 409

push() (IPython.parallel.client.view.DirectView
 method), 760

push_accepts_more()
 (IPython.core.inputsplitter.InputSplitter
 method), 372

push_accepts_more()
 (IPython.core.inputsplitter.IPythonInputSplitter
 method), 370

py_file_finder (class in IPython.testing.globalipapp), 834

pycmd2argv() (in module IPython.utils.process), 885

PyFileConfigLoader (class in IPython.config.loader), 283

pyfunc() (in module IPython.testing.plugin.dtexample), 841

pyfunc() (in module IPython.testing.plugin.simple), 843

pylab_activate() (in module IPython.lib.pylabtools), 586

PyPromptTransformer (class in IPython.core.prefilter), 505

python_func_kw_matches()
 (IPython.core.completer.IPCCompleter
 method), 302

python_matches() (IPython.core.completer.IPCCompleter
 method), 302

PythonOpsChecker (class in IPython.core.prefilter), 506

PythonRunner (class in IPython.lib.iprunner), 576

PYTHONSTARTUP, 943

Q

query (IPython.parallel.controller.hub.Hub attribute), 790

queue (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 641

queue (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 647

queue (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 651

queue (IPython.parallel.apps.launcher.LSFLauncher attribute), 654

queue (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 678

queue (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 681

queue (IPython.parallel.apps.launcher.PBSLauncher attribute), 685

queue (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 689

queue (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 692

queue (IPython.parallel.apps.launcher.SGELauncher attribute), 696

queue (IPython.parallel.controller.hub.EngineConnector attribute), 785

queue_reEXP (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 641

queue_reEXP (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 648

queue_reEXP (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 651

queue_reEXP (IPython.parallel.apps.launcher.LSFLauncher attribute), 654

queue_reEXP (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 678

queue_reEXP (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 682

queue_reEXP (IPython.parallel.apps.launcher.PBSLauncher attribute), 685

queue_reEXP (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 689

queue_reEXP (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 692

queue_reEXP (IPython.parallel.apps.launcher.SGELauncher attribute), 696

queue_status() (IPython.parallel.client.Client
 method), 749

queue_status() (IPython.parallel.client.view.DirectView.aise_ exception() (IPython.parallel.error.CompositeError method), 760
method), 813
queue_status() (IPython.parallel.client.view.LoadBalancedView.ainViewwall() (in module IPython.testing.plugin.dtexample), 841
method), 765
queue_status() (IPython.parallel.client.view.View ranfunc() (in module IPython.testing.plugin.dtexample), 842
method), 770
queue_status() (IPython.parallel.controller.hub.Hub raw_input_ext() (in module IPython.utils.io), 866
method), 790
queue_status() (IPython.parallel.controller.hub.Hub raw_input_multi() (in module IPython.utils.io), 866
method), 790
queue_template (IPython.parallel.apps.launcher.BatchSystem.ainplate() (in module IPython.utils.io), 866
attribute), 641
raw_print_err() (in module IPython.utils.io), 866
queue_template (IPython.parallel.apps.launcher.LSFControllerLainplate(IPython.lib.demo.ClearDemo attribute),
attribute), 648
558
queue_template (IPython.parallel.apps.launcher.LSFEngineSetLainplate(IPython.lib.demo.ClearIPDemo attribute),
attribute), 651
560
queue_template (IPython.parallel.apps.launcher.LSFLauncher re_auto (IPython.lib.demo.IPythonDemo attribute),
attribute), 655
re_auto (IPython.lib.demo.IPythonLineDemo attribute),
queue_template (IPython.parallel.apps.launcher.PBSControllerLainplate(IPython.lib.demo.IPythonLineDemo attribute),
attribute), 678
564
queue_template (IPython.parallel.apps.launcher.PBSEngineSetLainplate(IPython.lib.demo.IPythonLineDemo attribute),
attribute), 682
566
queue_template (IPython.parallel.apps.launcher.PBSLauncher re_auto_all (IPython.lib.demo.ClearDemo attribute),
attribute), 685
558
queue_template (IPython.parallel.apps.launcher.SGECControllerLainplate(IPython.lib.demo.ClearIPDemo attribute),
attribute), 689
560
queue_template (IPython.parallel.apps.launcher.SGEEngineSetLainplate(IPython.lib.demo.Demo attribute), 562
attribute), 692
re_auto_all (IPython.lib.demo.IPythonDemo attribute),
queue_template (IPython.parallel.apps.launcher.SGELauncher tribute), 564
attribute), 696
re_auto_all (IPython.lib.demo.IPythonLineDemo attribute),
QueueCleared (class in IPython.parallel.error), 819
tribute), 566
queues (IPython.parallel.controller.hub.Hub attribute), 790
re_auto_all (IPython.lib.demo.LineDemo attribute),
quick_completer() (in module IPython.core.completerlib), 303
567
quiet (IPython.core.interactiveshell.InteractiveShell re_mark() (in module IPython.lib.demo), 568
attribute), 409
re_silent (IPython.lib.demo.ClearDemo attribute),
quiet() (IPython.core.displayhook.DisplayHook 558
method), 320
re_silent (IPython.lib.demo.ClearIPDemo attribute),
qw() (in module IPython.utils.text), 898
560
re_silent (IPython.lib.demo.Demo attribute), 562
qw_lol() (in module IPython.utils.text), 899
re_silent (IPython.lib.demo.IPythonDemo attribute),
qwflat() (in module IPython.utils.text), 899
564
re_silent (IPython.lib.demo.IPythonLineDemo attribute),
R 566
re_silent (IPython.lib.demo.LineDemo attribute),
r (IPython.parallel.client.asyncresult.AsyncHubResult 567
attribute), 740
re_stop (IPython.lib.demo.ClearDemo attribute),
r (IPython.parallel.client.asyncresult.AsyncMapResult 558
attribute), 741
re_stop (IPython.lib.demo.ClearIPDemo attribute),
r (IPython.parallel.client.asyncresult.AsyncResult 560
attribute), 742
re_stop (IPython.lib.demo.Demo attribute), 562

re_stop (IPython.lib.demo.IPythonDemo attribute), [564](#)

re_stop (IPython.lib.demo.IPythonLineDemo attribute), [566](#)

re_stop (IPython.lib.demo.LineDemo attribute), [567](#)

read_payload() (IPython.core.payload.PayloadManager method), [455](#)

readline_merge_completions (IPython.core.interactiveshell.InteractiveShell attribute), [409](#)

readline OMIT NAMES (IPython.core.interactiveshell.InteractiveShell attribute), [409](#)

readline_parse_and_bind (IPython.core.interactiveshell.InteractiveShell attribute), [410](#)

readline_remove_delims (IPython.core.interactiveshell.InteractiveShell attribute), [410](#)

readline_use (IPython.core.interactiveshell.InteractiveShell attribute), [410](#)

ReadlineNoRecord (class in IPython.core.interactiveshell), [415](#)

ready() (AsyncResult method), [169](#)

ready() (IPython.parallel.client.asyncresult.AsyncHubResult method), [740](#)

ready() (IPython.parallel.client.asyncresult.AsyncMapResult method), [741](#)

ready() (IPython.parallel.client.asyncresult.AsyncResult method), [742](#)

real_name() (in module IPython.core.magic_arguments), [448](#)

rebindFunctionGlobals() (in module IPython.utils.pickleutil), [883](#)

Red (IPython.utils.coloransi.InputTermColors attribute), [857](#)

Red (IPython.utils.coloransi.TermColors attribute), [858](#)

reduce_code() (in module IPython.utils.codeutil), [854](#)

Reference (class in IPython.utils.pickleutil), [883](#)

refill_readline_hist() (IPython.core.interactiveshell.InteractiveShell method), [410](#)

register() (IPython.config.loader.ArgumentParser method), [278](#)

register() (IPython.core.magic_arguments.MagicArgumentParser method), [446](#)

register() (IPython.parallel.engine.EngineFactory method), [805](#)

register_checker() (IPython.core.prefilter.PrefilterManager method), [502](#)

register_engine() (IPython.parallel.controller.hub.Hub method), [790](#)

register_handler() (IPython.core.prefilter.PrefilterManager method), [502](#)

register_post_execute() (IPython.core.interactiveshell.InteractiveShell method), [410](#)

register_transformer() (IPython.core.prefilter.PrefilterManager method), [502](#)

registrar (IPython.parallel.engine.EngineFactory attribute), [805](#)

registration (IPython.parallel.controller.hub.EngineConnector attribute), [785](#)

registration_timeout

RegistrationFactory (class in IPython.parallel.controller.hub.Hub attribute), [790](#)

regport (IPython.parallel.factory), [823](#)

regport (IPython.parallel.controller.hub.HubFactory attribute), [793](#)

regport (IPython.parallel.engine.EngineFactory attribute), [805](#)

regport (IPython.parallel.factory.RegistrationFactory attribute), [825](#)

reinit_logging() (IPython.parallel.apps.baseapp.BaseParallelApplication method), [591](#)

reinit_logging() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), [602](#)

reinit_logging() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), [608](#)

reinit_logging() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), [613](#)

reinit_logging() (IPython.parallel.apps.ipcontrollerapp.IPControllerApplication method), [620](#)

reinit_logging() (IPython.parallel.apps.ipengineapp.IPEngineApp method), [626](#)

reinit_logging() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), [634](#)

rekey() (in module IPython.utils.jsonutil), [872](#)

reload() (IPython.lib.deepreload.ModulePlaceholder method), [554](#)

reload() (IPython.lib.demo.ClearDemo method), [558](#)

reload() (IPython.lib.demo.ClearIPDemo method), 560
reload() (IPython.lib.demo.Demo method), 562
reload() (IPython.lib.demo.IPythonDemo method), 564
reload() (IPython.lib.demo.IPythonLineDemo method), 566
reload() (IPython.lib.demo.LineDemo method), 568
reload_extension() (IPython.core.extensions.ExtensionManager method), 329
remote() (in module IPython.parallel.client.remotefunction), 756
remote() (IPython.parallel.client.view.DirectView method), 760
remote() (IPython.parallel.client.view.LoadBalancedView method), 765
remote() (IPython.parallel.client.view.View method), 770
RemoteError (class in IPython.parallel.error), 820
RemoteFunction (class in IPython.parallel.client.remotefunction), 755
remove (IPython.parallel.controller.dependency.Dependency attribute), 774
remove (IPython.utils.text.SList attribute), 896
remove() (IPython.lib.backgroundjobs.BackgroundJob method), 552
remove() (IPython.lib.pretty.GroupQueue method), 582
remove_all_observers() (IPython.utils.notification.NotificationCenter method), 877
remove_builtin() (IPython.core.builtin_trap.BuiltinTrap method), 295
remove_comments() (in module IPython.core.inputsplitter), 373
remove_pid_file() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 591
remove_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 602
remove_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterStart class in IPython.utils.autoattr), 853
remove_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 783
remove_pid_file() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp method), 620
remove_pid_file() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 807
method), 626
remove_pid_file() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 634
replace (IPython.utils.text.LSString attribute), 893
report() (in module IPython.testing.iptest), 836
repr_type() (in module IPython.utils.traits), 935
RepresentationPrinter (class in IPython.lib.pretty), 583
Managed_nodes (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 724
requested_nodes (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 730
requested_nodes (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 735
require() (in module IPython.parallel.controller.dependency), 775
reset (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 608
reset() (IPython.core.debugger.Pdb method), 313
reset() (IPython.core.history.HistoryManager method), 362
reset() (IPython.core.inputsplitter.InputSplitter method), 372
reset() (IPython.core.inputsplitter.IPythonInputSplitter method), 370
reset() (IPython.core.interactiveshell.InteractiveShell method), 410
reset() (IPython.lib.demo.ClearDemo method), 558
reset() (IPython.lib.demo.ClearIPDemo method), 560
reset() (IPython.lib.demo.Demo method), 562
reset() (IPython.lib.demo.IPythonDemo method), 564
reset() (IPython.lib.demo.IPythonLineDemo method), 566
reset() (IPython.lib.demo.LineDemo method), 568
ParallelApplication (IPython.utils.autoattr.ResetMixin method), 853
RestartEngine (IPython.core.interactiveshell.InteractiveShell method), 410
ResetMixin (class in IPython.utils.autoattr), 853
responses (IPython.parallel.controller.heartmonitor.HeartMonitor method), 410
stop(), 783
restart_kernel() (IPython.parallel.engine.kernelstarter.KernelStarter method), 807
method), 807
restore_sys_module_state()
APython.core.interactiveshell.InteractiveShell

method), 410
 resubmit (IPython.parallel.controller.hub.Hub attribute), 790
 resubmit() (IPython.parallel.client.client.Client method), 749
 resubmit_task() (IPython.parallel.controller.hub.Hub method), 790
 result (IPython.parallel.client.asyncresult.AsyncHubResult attribute), 740
 result (IPython.parallel.client.asyncresult.AsyncMapResult attribute), 741
 result (IPython.parallel.client.asyncresult.AsyncResult attribute), 742
 result() (IPython.lib.backgroundjobs.BackgroundJobManager attribute), 552
 result_dict (IPython.parallel.client.asyncresult.AsyncHubResult attribute), 740
 result_dict (IPython.parallel.client.asyncresult.AsyncMapResult attribute), 741
 result_dict (IPython.parallel.client.asyncresult.AsyncResult attribute), 743
 result_status() (IPython.parallel.client.client.Client method), 750
 ResultAlreadyRetrieved (class in IPython.parallel.error), 820
 ResultNotCompleted (class in IPython.parallel.error), 820
 results (IPython.parallel.client.client.Client attribute), 750
 results (IPython.parallel.client.view.DirectView attribute), 760
 results (IPython.parallel.client.view.LoadBalancedView attribute), 765
 results (IPython.parallel.client.view.View attribute), 770
 resume_receiving() (IPython.parallel.controller.scheduler.TaskScheduler method), 798
 retries (IPython.parallel.client.view.LoadBalancedView attribute), 765
 retries (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798
 reuse_files (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 620
 reverse (IPython.utils.text.SList attribute), 896
 ReverseDict (class in IPython.parallel.util), 827
 rewrite (IPython.core.autocall.ExitAutocall attribute), 293
 rewrite (IPython.core.autocall.IPyAutocall attribute), 293
 tribute), 293
 rewrite (IPython.core.autocall.ZMQExitAutocall attribute), 293
 rfind (IPython.utils.text.LSString attribute), 893
 rindex (IPython.utils.text.LSString attribute), 893
 rjust (IPython.utils.text.LSString attribute), 893
 rlcomplete() (IPython.core.completer.IPCompleter method), 302
 RoundRobinMap (class in IPython.parallel.client.map), 753
 rpartition (IPython.utils.text.LSString attribute), 893
 rsplit (IPython.utils.text.LSString attribute), 894
 rstrip (IPython.utils.text.LSString attribute), 894
 run() (IPython.core.debugger.Pdb method), 313
 run() (IPython.core.history.HistorySavingThread method), 363
 run() (IPython.lib.backgroundjobs.BackgroundJobBase method), 548
 run() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 549
 run() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 550
 run() (IPython.parallel.apps.win32support.ForwarderThread method), 721
 run() (IPython.parallel.client.view.DirectView method), 760
 run() (IPython.testing.ipctest.IPTester method), 835
 run_ast_nodes() (IPython.core.interactiveshell.InteractiveShell method), 410
 run_cell() (IPython.core.interactiveshell.InteractiveShell method), 410
 run_cell() (IPython.lib.demo.ClearDemo method), 558
 run_cell() (IPython.lib.demo.ClearIPDemo method), 568
 run_cell() (IPython.lib.demo.Demo method), 562
 run_cell() (IPython.lib.demo.IPythonDemo method), 564
 run_cell() (IPython.lib.demo.IPythonLineDemo method), 566
 run_code() (IPython.core.interactiveshell.InteractiveShell method), 411
 run_file() (IPython.lib.irunner.InteractiveRunner method), 576
 run_file() (IPython.lib.irunner.IPythonRunner

method), 574
run_file() (IPython.lib.irunner.PythonRunner method), 576
run_file() (IPython.lib.irunner.SAGERunner method), 578
run_ipctest() (in module IPython.testing.ipctest), 836
run_ipctestall() (in module IPython.testing.ipctest), 836
run_source() (IPython.lib.irunner.InteractiveRunner method), 576
run_source() (IPython.lib.irunner.IPythonRunner method), 574
run_source() (IPython.lib.irunner.PythonRunner method), 577
run_source() (IPython.lib.irunner.SAGERunner method), 578
run_until_canceled(IPython.parallel.apps.winhpcjob.IPCController attribute), 689
run_until_canceled(IPython.parallel.apps.winhpcjob.IPEngineSet attribute), 692
run_until_canceled(IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 696
runcall() (IPython.core.debugger.Pdb method), 313
runcode() (IPython.core.interactiveshell.InteractiveShell method), 411
runctx() (IPython.core.debugger.Pdb method), 313
runeval() (IPython.core.debugger.Pdb method), 313
runner (IPython.testing.ipctest.IPTester attribute), 835
RunnerFactory (class in IPython.lib.irunner), 577
RunnerFactory (class in IPython.testing.mkdoctests), 839
running (IPython.parallel.apps.launcher.BaseLauncher attribute), 638
running (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 641
running (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 644
running (IPython.parallel.apps.launcher.LocalController attribute), 658
running (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 661
running (IPython.parallel.apps.launcher.LocalEngineSet attribute), 663
running (IPython.parallel.apps.launcher.LocalProcessLauncher attribute), 666
running (IPython.parallel.apps.launcher.LSFController attribute), 648
running (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 651
running (IPython.parallel.apps.launcher.LSFLauncher attribute), 655
running (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 669
running (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher attribute), 672
running (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 675
running (IPython.parallel.apps.launcher.PBSCControllerLauncher attribute), 678
running (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 682
running (IPython.parallel.apps.launcher.PBSLauncher attribute), 685
running (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 689
running (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 692
running (IPython.parallel.apps.launcher.SGELauncher attribute), 696
running (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 699
running (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 702
running (IPython.parallel.apps.launcher.SSHEngineSetLauncher attribute), 705
running (IPython.parallel.apps.launcher.SSHLauncher attribute), 708
in running (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher attribute), 711
running (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher attribute), 714
running (IPython.parallel.apps.launcher.WindowsHPCLauncher attribute), 716
S
IPython.utils.text.LSString attribute), 894
s (IPython.utils.text.SList attribute), 896
sumatches() (IPython.utils.strdispatch.StrDispatch method), 886
Ifaunusfile() (IPython.core.interactiveshell.InteractiveShell method), 411
safe_execfile_ipy() (IPython.core.interactiveshell.InteractiveShell method), 411
SAGERunner (class in IPython.lib.irunner), 577
save_connection_dict()
IPython.parallel.apps.ipcontrollerapp.IPCControllerApp

method), 620
`save_flag` (IPython.core.history.HistoryManager attribute), 362
`save_ids()` (in module IPython.parallel.client.view), 771
`save_iopub_message()`
 (IPython.parallel.controller.hub.Hub method), 790
`save_queue_request()`
 (IPython.parallel.controller.hub.Hub method), 790
`save_queue_result()` (IPython.parallel.controller.hub.Hub method), 790
`save_sys_module_state()`
 (IPython.core.interactiveshell.InteractiveShell seek() (IPython.lib.demo.LineDemo method), 568
 method), 412
`save_task_destination()`
 (IPython.parallel.controller.hub.Hub method), 790
`save_task_request()` (IPython.parallel.controller.hub.Hub method), 790
`save_task_result()` (IPython.parallel.controller.hub.Hub method), 790
`save_thread` (IPython.core.history.HistoryManager attribute), 362
`save_unmet()` (IPython.parallel.controller.scheduler.TaskScheduler attribute), 741
 method), 798
`save_urls()` (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 743
 method), 620
`scatter()` (IPython.parallel.client.view.DirectView method), 760
`scheduler` (IPython.parallel.apps.launcher.WindowsHPCController attribute), 712
 attribute), 711
`scheduler` (IPython.parallel.apps.launcher.WindowsHPCEngine attribute), 712
 attribute), 714
`scheduler` (IPython.parallel.apps.launcher.WindowsHPCLaunch attribute), 716
 attribute), 716
`scheme` (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798
`scheme_name` (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798
`search()` (IPython.core.history.HistoryManager method), 362
`section_sep` (IPython.core.crashhandler.CrashHandler attribute), 305
`section_sep` (IPython.parallel.apps.baseapp.ParallelCrashHandler attribute), 593
`secure` (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 777
`attribute)`, 620
`security_dir` (IPython.core.profiledir.ProfileDir attribute), 528
`security_dir_name` (IPython.core.profiledir.ProfileDir attribute), 528
`SecurityError` (class in IPython.parallel.error), 820
`seek()` (IPython.lib.demo.ClearDemo method), 558
`seek()` (IPython.lib.demo.ClearIPDemo method), 560
`seek()` (IPython.lib.demo.Demo method), 562
`seek()` (IPython.lib.demo.IPythonDemo method), 564
`seek()` (IPython.lib.demo.IPythonLineDemo method), 566
`select_seek()` (IPython.lib.demo.LineDemo method), 568
`select_figure_format()` (in module IPython.lib.pylabtools), 586
`select_random_ports()` (in module IPython.parallel.util), 829
`send_apply_message()`
 (IPython.parallel.client.client.Client method), 750
`sent` (IPython.parallel.client.asyncresult.AsyncHubResult attribute), 740
`sent` (IPython.parallel.client.asyncresult.AsyncMapResult attribute), 741
`sent` (IPython.parallel.client.asyncresult.AsyncResult attribute), 743
`separate_in` (IPython.core.interactiveshell.InteractiveShell attribute), 412
`separate_out` (IPython.core.interactiveshell.InteractiveShell attribute), 412
`separate_out2` (IPython.core.interactiveshell.InteractiveShell attribute), 412
`SeparateUnicode` (class in IPython.core.interactiveshell), 416
`SerializationError` (class in IPython.parallel.error), 821
`SerializationError` (class in IPython.utils.newserialized), 874
`serialize()` (in module IPython.utils.newserialized), 875
`serialize_object()` (in module IPython.parallel.util), 829
`Serialized` (class in IPython.utils.newserialized), 874
`ShaHash` (class in IPython.utils.newserialized), 874
`session` (IPython.parallel.controller.dictdb.BaseDB attribute), 777

session (IPython.parallel.controller.dictdb.DictDB attribute), 780

session (IPython.parallel.controller.hub.Hub attribute), 790

session (IPython.parallel.controller.hub.HubFactory attribute), 793

session (IPython.parallel.controller.scheduler.TaskScheduler attribute), 798

session (IPython.parallel.controller.sqlitedb.SQLiteDB attribute), 802

session (IPython.parallel.engine.engine.EngineFactory attribute), 805

session (IPython.parallel.engine.streamkernel.Kernel attribute), 810

session (IPython.parallel.factory.RegistrationFactory attribute), 825

session_number (IPython.core.history.HistoryManager attribute), 362

Set (class in IPython.utils.traits), 926

set() (IPython.core.display_trap.DisplayTrap method), 318

set_active_scheme() (IPython.core.oinspect.Inspector method), 451

set_active_scheme() (IPython.utils.coloransi.ColorSchemeTable method), 856

set_autoindent() (IPython.core.interactiveshell.InteractiveShell method), 412

set_break() (IPython.core.debugger.Pdb method), 313

set_colors() (IPython.core.debugger.Pdb method), 313

set_colors() (IPython.core.displayhook.DisplayHook method), 321

set_colors() (IPython.core.prompts.Prompt1 method), 531

set_colors() (IPython.core.prompts.Prompt2 method), 531

set_colors() (IPython.core.prompts.PromptOut method), 532

set_colors() (IPython.core.ultratb.AutoFormattedTB method), 538

set_colors() (IPython.core.ultratb.ColorTB method), 540

set_colors() (IPython.core.ultratb.FormattedTB method), 541

set_colors() (IPython.core.ultratb.ListTB method), 542

set_colors() (IPython.core.ultratb.SyntaxTB method), 544

set_colors() (IPython.core.ultratb.TBTools method), 545

set_colors() (IPython.core.ultratb.VerboseTB method), 546

set_completer_frame()

set_continue() (IPython.core.debugger.Pdb method), 313

set_custom_completer() (IPython.core.interactiveshell.InteractiveShell method), 412

set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 412

set_default_value() (IPython.core.interactiveshell.SeparateUnicode method), 416

set_default_value() (IPython.utils.traits.Any method), 902

set_default_value() (IPython.utils.traits.Bool method), 902

set_default_value() (IPython.utils.traits.Bytes method), 903

set_default_value() (IPython.utils.traits.CaselessStrEnum method), 911

set_default_value() (IPython.utils.traits.CBool method), 904

set_default_value() (IPython.utils.traits.CBytes method), 905

set_default_value() (IPython.utils.traits.CComplex method), 906

set_default_value() (IPython.utils.traits.CFloat method), 907

set_default_value() (IPython.utils.traits.CInt method), 908

set_default_value() (IPython.utils.traits.ClassBasedTraitType method), 912

set_default_value() (IPython.utils.traits.CLong method), 909

set_default_value() (IPython.utils.traits.Complex method), 913

set_default_value() (IPython.utils.traits.Container method), 915

set_default_value() (IPython.utils.traits.CUnicode method), 910

set_default_value() (IPython.utils.traits.Dict

method), 916
set_default_value() (IPython.utils.traits.DottedObject~~Name~~) (IPython.core.autocall.IPyAutocall method),
 method), 917
set_default_value() (IPython.utils.traits.Enum method), 918
set_default_value() (IPython.utils.traits.Float method), 919
set_default_value() (IPython.utils.traits.Instance method), 921
set_default_value() (IPython.utils.traits.Int method), 922
set_default_value() (IPython.utils.traits.List method), 923
set_default_value() (IPython.utils.traits.Long method), 924
set_default_value() (IPython.utils.traits.ObjectNameset_metadata() (IPython.utils.traits.CaselessStrEnum method), 926
 method), 926
set_default_value() (IPython.utils.traits.Set method), 927
set_default_value() (IPython.utils.traits.TCPAddressset_metadata() (IPython.utils.traits.CComplex method), 928
 method), 928
set_default_value() (IPython.utils.traits.This method), 929
set_default_value() (IPython.utils.traits.TraitType method), 930
set_default_value() (IPython.utils.traits.Tuple method), 932
set_default_value() (IPython.utils.traits.Type method), 933
set_default_value() (IPython.utils.traits.Unicode method), 934
set_defaults() (IPython.config.loader.ArgumentParser set_metadata() (IPython.utils.traits.Container method), 278
 method), 278
set_defaults() (IPython.core.magic_arguments.MagicArgumentParser set_metadata() (IPython.utils.traits.CUnicode method), 447
 method), 447
set_delims() (IPython.core.completer.CompletionSplitter set_metadata() (IPython.utils.traits.Dict method),
 method), 300
set_flags() (IPython.parallel.client.view.DirectView set_metadata() (IPython.utils.traits.DottedObjectName method), 761
 method), 761
set_flags() (IPython.parallel.client.view.LoadBalancedView set_metadata() (IPython.utils.traits.Enum method), 765
 method), 765
set_flags() (IPython.parallel.client.view.View set_metadata() (IPython.utils.traits.Float method),
 method), 770
set_hook() (IPython.core.interactiveshell.InteractiveShell set_metadata() (IPython.utils.traits.Instance method), 413
 method), 413
set_inpthook() (IPython.lib.inpthook.InputHookManager set_metadata() (IPython.utils.traits.Int method),
 method), 572
set_ip() (IPython.core.autocall.ExitAutocall set_metadata() (IPython.utils.traits.List method),
 method), 293
set_ip() (IPython.core.autocall.ZMQExitAutocall method), 293
set_metadata() (IPython.core.interactiveshell.SeparateUnicode method), 416
set_metadata() (IPython.utils.traits.Any method), 902
set_metadata() (IPython.utils.traits.Bool method), 903
set_metadata() (IPython.utils.traits.Bytes method), 903
set_metadata() (IPython.utils.traits.CaselessStrEnum method), 911
set_metadata() (IPython.utils.traits.CBool method), 904
set_metadata() (IPython.utils.traits.CBytes method), 905
set_metadata() (IPython.utils.traits.CComplex method), 906
set_metadata() (IPython.utils.traits.CFloat method), 907
set_metadata() (IPython.utils.traits.CInt method), 908
set_metadata() (IPython.utils.traits.ClassBasedTraitType method), 912
set_metadata() (IPython.utils.traits.CLong method), 909
set_metadata() (IPython.utils.traits.Complex method), 913
set_metadata() (IPython.utils.traits.Container method), 915
set_metadata() (IPython.utils.traits.CUnicode method), 910
set_metadata() (IPython.utils.traits.Dict method), 916
set_metadata() (IPython.utils.traits.DottedObjectName method), 917
set_metadata() (IPython.utils.traits.Enum method), 918
set_metadata() (IPython.utils.traits.Float method), 919
set_metadata() (IPython.utils.traits.Instance method), 921
set_metadata() (IPython.utils.traits.Int method), 922
set_metadata() (IPython.utils.traits.List method), 923

923
set_metadata() (IPython.utils.traits.Long method),
 924
set_metadata() (IPython.utils.traits.ObjectName
 method), 926
set_metadata() (IPython.utils.traits.Set method),
 927
set_metadata() (IPython.utils.traits.TCPAddress
 method), 928
set_metadata() (IPython.utils.traits.This method),
 929
set_metadata() (IPython.utils.traits.TraitType
 method), 931
set_metadata() (IPython.utils.traits.Tuple
 method), 932
set_metadata() (IPython.utils.traits.Type method),
 933
set_metadata() (IPython.utils.traits.Unicode
 method), 934
set_mode() (IPython.core.ultratb.AutoFormattedTB
 method), 538
set_mode() (IPython.core.ultratb.ColorTB method),
 540
set_mode() (IPython.core.ultratb.FormattedTB
 method), 541
set_next() (IPython.core.debugger.Pdb method), 313
set_next_input() (IPython.core.interactiveshell.InteractiveShell
 method), 413
set_p_str() (IPython.core.prompts.BasePrompt
 method), 530
set_p_str() (IPython.core.prompts.Prompt1 method),
 531
set_p_str() (IPython.core.prompts.Prompt2 method),
 531
set_p_str() (IPython.core.prompts.PromptOut
 method), 532
set_quit() (IPython.core.debugger.Pdb method), 313
set_readline_completer()
 (IPython.core.interactiveshell.InteractiveShell
 method), 413
set_return() (IPython.core.debugger.Pdb method),
 313
set_step() (IPython.core.debugger.Pdb method), 313
set_term_title() (in module IPython.utils.terminal),
 888
set_trace() (IPython.core.debugger.Pdb method),
 313
set_until() (IPython.core.debugger.Pdb method), 313

setDaemon() (IPython.core.history.HistorySavingThread
 method), 364
setDaemon() (IPython.lib.backgroundjobs.BackgroundJobBase
 method), 548
setDaemon() (IPython.lib.backgroundjobs.BackgroundJobExpr
 method), 549
setDaemon() (IPython.lib.backgroundjobs.BackgroundJobFunc
 method), 550
setDaemon() (IPython.parallel.apps.win32support.ForwarderThread
 method), 721
setdefault (IPython.config.loader.Config attribute),
 279
setdefault (IPython.parallel.client.client.Metadata at-
 tribute), 752
setdefault (IPython.parallel.util.Namespace at-
 tribute), 827
setdefault (IPython.parallel.util.ReverseDict at-
 tribute), 828
setdefault (IPython.testing.globalipapp.ipnsdict at-
 tribute), 834
setdefault (IPython.utils.coloransi.ColorSchemeTable
 attribute), 856
setdefault (IPython.utils.ipstruct.Struct attribute),
 871
setdefault() (IPython.utils.pickleshare.PickleShareDB
 method), 881
setHandle() (IPython.core.history.HistorySavingThread
 method), 364

setName() (IPython.lib.backgroundjobs.BackgroundJobBase
 method), 548
setName() (IPython.lib.backgroundjobs.BackgroundJobExpr
 method), 549
setName() (IPython.lib.backgroundjobs.BackgroundJobFunc
 method), 550
setName() (IPython.parallel.apps.win32support.ForwarderThread
 method), 721

setup() (IPython.core.debugger.Pdb method), 313
SGEControllerLauncher (class in
 IPython.parallel.apps.launcher), 686
SGEEngineSetLauncher (class in
 IPython.parallel.apps.launcher), 690
SGELauncher (class in
 IPython.parallel.apps.launcher), 693
shell (IPython.core.alias.AliasManager attribute),
 286
shell (IPython.core.builtin_trap.BuiltinTrap at-
 tribute), 295
shell (IPython.core.displayhook.DisplayHook

attribute), 321
shell (IPython.core.extensions.ExtensionManager attribute), 329
shell (IPython.core.history.HistoryManager attribute), 362
shell (IPython.core.prefilter.AliasChecker attribute), 463
shell (IPython.core.prefilter.AliasHandler attribute), 465
shell (IPython.core.prefilter.AssignMagicTransformer attribute), 467
shell (IPython.core.prefilter.AssignmentChecker attribute), 470
shell (IPython.core.prefilter.AssignSystemTransformer attribute), 468
shell (IPython.core.prefilter.AutocallChecker attribute), 476
shell (IPython.core.prefilter.AutoHandler attribute), 472
shell (IPython.core.prefilter.AutoMagicChecker attribute), 474
shell (IPython.core.prefilter.EmacsChecker attribute), 478
shell (IPython.core.prefilter.EmacsHandler attribute), 479
shell (IPython.core.prefilter.EscCharsChecker attribute), 481
shell (IPython.core.prefilter.HelpHandler attribute), 483
shell (IPython.core.prefilter.IPyAutocallChecker attribute), 485
shell (IPython.core.prefilter.IPyPromptTransformer attribute), 487
shell (IPython.core.prefilter.MacroChecker attribute), 489
shell (IPython.core.prefilter.MacroHandler attribute), 491
shell (IPython.core.prefilter.MagicHandler attribute), 493
shell (IPython.core.prefilter.MultiLineMagicChecker attribute), 495
shell (IPython.core.prefilter.PrefilterChecker attribute), 497
shell (IPython.core.prefilter.PrefilterHandler attribute), 499
shell (IPython.core.prefilter.PrefilterManager attribute), 502
shell (IPython.core.prefilter.PrefilterTransformer attribute), 504
shell (IPython.core.prefilter.PyPromptTransformer attribute), 506
shell (IPython.core.prefilter.PythonOpsChecker attribute), 508
shell (IPython.core.prefilter.ShellEscapeChecker attribute), 510
shell (IPython.core.prefilter.ShellEscapeHandler attribute), 511
shell (IPython.core.shellapp.InteractiveShellApp attribute), 535
shell_handlers (IPython.parallel.engine.streamkernel.Kernel attribute), 811
ShellEscapeChecker (class in IPython.core.prefilter), 508
ShellEscapeHandler (class in IPython.core.prefilter), 510
shlex_split() (in module IPython.core.completerlib), 304
show() (IPython.lib.demo.ClearDemo method), 558
show() (IPython.lib.demo.ClearIPDemo method), 560
show() (IPython.lib.demo.Demo method), 562
show() (IPython.lib.demo.IPythonDemo method), 564
show() (IPython.lib.demo.IPythonLineDemo method), 566
show() (IPython.lib.demo.LineDemo method), 568
show_all() (IPython.lib.demo.ClearDemo method), 558
show_all() (IPython.lib.demo.ClearIPDemo method), 560
show_all() (IPython.lib.demo.Demo method), 562
show_all() (IPython.lib.demo.IPythonDemo method), 564
show_all() (IPython.lib.demo.IPythonLineDemo method), 566
show_all() (IPython.lib.demo.LineDemo method), 568
show_exception_only() (IPython.core.ultratb.AutoFormattedTB method), 538
show_exception_only() (IPython.core.ultratb.ColorTB method), 540
show_exception_only()

(IPython.core.ultratb.FormattedTB
method), 541

show_exception_only() (IPython.core.ultratb.ListTB
method), 543

show_exception_only()
(IPython.core.ultratb.SyntaxTB method), 544

show_hidden() (in module IPython.utils.wildcard), 937

show_in_pager() (in module IPython.core.hooks), 368

show_usage() (IPython.core.interactiveshell.InteractiveShell
method), 413

showdiff() (in module IPython.utils.upgradedir), 935

showindentationerror()
(IPython.core.interactiveshell.InteractiveShell
method), 413

showsyntaxerror() (IPython.core.interactiveshell.InteractiveShell
method), 413

showtraceback() (IPython.core.interactiveshell.InteractiveShell
method), 413

shutdown() (IPython.parallel.client.client.Client
method), 750

shutdown() (IPython.parallel.client.view.DirectView
method), 761

shutdown() (IPython.parallel.client.view.LoadBalancedView
method), 766

shutdown() (IPython.parallel.client.view.View
method), 770

shutdown_hook() (in module IPython.core.hooks), 368

shutdown_kernel() (IPython.parallel.engine.kernelstarters.Kernel
method), 807

shutdown_request() (IPython.parallel.controller.hub.Hub
method), 790

shutdown_request() (IPython.parallel.engine.kernelstarters.Kernel
method), 807

shutdown_request() (IPython.parallel.engine.streamkernel.Kernel
method), 811

sigint_handler() (IPython.parallel.apps.ipclusterapp.IPClusterSigIntHandler
method), 602

sigint_handler() (IPython.parallel.apps.ipclusterapp.IPClusterSigIntHandler
method), 608

signal (IPython.parallel.apps.ipclusterapp.IPClusterStopSignal
attribute), 613

signal() (IPython.parallel.apps.launcher.BaseLauncher
method), 638

signal() (IPython.parallel.apps.launcher.BatchSystemLauncher
method), 641

signal() (IPython.parallel.apps.launcher.IPCLauncher
method), 644

signal() (IPython.parallel.apps.launcher.LocalControllerLauncher
method), 658

signal() (IPython.parallel.apps.launcher.LocalEngineLauncher
method), 661

signal() (IPython.parallel.apps.launcher.LocalEngineSetLauncher
method), 663

signal() (IPython.parallel.apps.launcher.LocalProcessLauncher
method), 666

signal() (IPython.parallel.apps.launcher.LSFControllerLauncher
method), 648

signal() (IPython.parallel.apps.launcher.LSFEngineSetLauncher
method), 651

signal() (IPython.parallel.apps.launcher.LSFLauncher
method), 655

signal() (IPython.parallel.apps.launcher.MPIExecControllerLauncher
method), 669

signal() (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher
method), 672

signal() (IPython.parallel.apps.launcher.MPIExecLauncher
method), 675

signal() (IPython.parallel.apps.launcher.PBSControllerLauncher
method), 678

signal() (IPython.parallel.apps.launcher.PBSEngineSetLauncher
method), 682

signal() (IPython.parallel.apps.launcher.PBSLauncher
method), 685

signal() (IPython.parallel.apps.launcher.SGEControllerLauncher
method), 689

signal() (IPython.parallel.apps.launcher.SGEEngineSetLauncher
method), 692

signal() (IPython.parallel.apps.launcher.SGELauncher
method), 696

signal() (IPython.parallel.apps.launcher.SSHControllerLauncher
method), 699

signal() (IPython.parallel.apps.launcher.SSHEngineLauncher
method), 702

signal() (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher
method), 711

signal() (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher
method), 714

signal() (IPython.parallel.apps.launcher.WindowsHPCLauncher
method), 716

method), 716
`signal_children()` (in module IPython.parallel.util), 829
`signal_kernel()` (IPython.parallel.engine.kernelstarter.KernelStarter(IPython.core.inputsplitter.IPythonInputSplitter method), 807
`single_dir_expand()` (in module IPython.core.completer), 303
`singleton_printers` (IPython.core.formatters.BaseFormatter), 333
`singleton_printers` (IPython.core.formatters.HTMLFormatter), 339
`singleton_printers` (IPython.core.formatters.JavascriptFormatter), 345
`singleton_printers` (IPython.core.formatters.JSONFormatter), 342
`singleton_printers` (IPython.core.formatters.LatexFormatter), 348
`singleton_printers` (IPython.core.formatters.PlainTextFormatter), 354
`singleton_printers` (IPython.core.formatters.PNGFormatter), 351
`singleton_printers` (IPython.core.formatters.SVGFormatter), 357
`SingletonConfigurable` (class in IPython.config.configurable), 273
`skip()` (in module IPython.testing.decorators), 831
`skip_doctest` (IPython.parallel.client.view.DirectView attribute), 761
`skip_doctest` (IPython.parallel.client.view.LoadBalancedView attribute), 766
`skip_doctest` (IPython.parallel.client.view.View attribute), 770
`skip_doctest()` (in module IPython.testing.skipdoctest), 845
`skipif()` (in module IPython.testing.decorators), 831
`SList` (class in IPython.utils.text), 895
`snip_print()` (in module IPython.core.page), 453
`soft_define_alias()` (IPython.core.alias.AliasManager method), 286
`softspace()` (in module IPython.core.interactiveshell), 417
`sort()` (IPython.utils.text.SList method), 897
`sort_checkers()` (IPython.core.prefilter.PrefilterManager method), 502
`sort_compare()` (in module IPython.utils.data), 859
`sort_transformers()` (IPython.core.prefilter.PrefilterManager method), 502
`source` (IPython.core.inputsplitter.InputSplitter at- tribute), 372
`source` (IPython.core.inputsplitter.IPythonInputSplitter attribute), 370
`source_raw_reset()` (IPython.core.inputsplitter.InputSplitter method), 370
`source_reset()` (IPython.core.inputsplitter.IPythonInputSplitter method), 372
`source_reset()` (IPython.core.inputsplitter.IPythonInputSplitter method), 370
`SpannableInput` (class in IPython.core.interactiveshell), 417
`split` (IPython.utils.text.LSString attribute), 894
`split_line()` (IPython.core.completer.CompletionSplitter method), 300
`split_viewurl()` (in module IPython.parallel.util), 829
`split_user_input()` (in module IPython.core.inputsplitter), 373
`split_user_input()` (in module IPython.core.splitinput), 535
`splitlines` (IPython.utils.text.LSString attribute), 894
`spstr` (IPython.utils.text.LSString attribute), 894
`spstr` (IPython.utils.text.SList attribute), 897
`SQLiteDB` (class in IPython.parallel.controller.sqlitedb), 800
`squash_dates()` (in module IPython.utils.jsonutil), 872
`ssh_args` (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 699
`ssh_args` (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 702
`ssh_args` (IPython.parallel.apps.launcher.SSHLauncher attribute), 708
`ssh_cmd` (IPython.parallel.apps.launcher.SSHControllerLauncher

attribute), 699
ssh_cmd (IPython.parallel.apps.launcher.SSHEngineLauncher.start() (IPython.core.profileapp.ProfileApp method),
attribute), 702 516
ssh_cmd (IPython.parallel.apps.launcher.SSHLauncher.start() (IPython.core.profileapp.ProfileCreate
attribute), 708 method), 520
ssh_server (IPython.parallel.apps.ipcontrollerapp.IPController.start() (IPython.core.profileapp.ProfileList method),
attribute), 620 524
SSHControllerLauncher (class in start() (IPython.lib.backgroundjobs.BackgroundJobBase
IPython.parallel.apps.launcher), 697 method), 548
SSHEngineLauncher (class in start() (IPython.lib.backgroundjobs.BackgroundJobExpr
IPython.parallel.apps.launcher), 700 method), 549
SSHEngineSetLauncher (class in start() (IPython.lib.backgroundjobs.BackgroundJobFunc
IPython.parallel.apps.launcher), 703 method), 550
SSHLauncher (class in start() (IPython.parallel.apps.baseapp.BaseParallelApplication
IPython.parallel.apps.launcher), 705 method), 592
stage_default_config_file() start() (IPython.parallel.apps.ipclusterapp.IPClusterApp
(IPython.core.application.BaseIPythonApplication method), 597
method), 291 start() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
method), 602
stage_default_config_file() start() (IPython.parallel.apps.ipclusterapp.IPClusterStart
(IPython.parallel.apps.baseapp.BaseParallelApplication method), 614
method), 591 start() (IPython.parallel.apps.ipclusterapp.IPClusterStop
method), 608
stage_default_config_file() start() (IPython.parallel.apps.ipclusterapp.IPClusterStart
(IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 614
method), 602 start() (IPython.parallel.apps.ipengineapp.IPEngineApp
method), 626
stage_default_config_file() start() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
(IPython.parallel.apps.ipclusterapp.IPClusterStart method), 634
method), 608 start() (IPython.parallel.apps.launcher.BaseLauncher
method), 638
stage_default_config_file() start() (IPython.parallel.apps.ipclusterapp.IPClusterStart
(IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 614
method), 613 start() (IPython.parallel.apps.launcher.BatchSystemLauncher
method), 642
stage_default_config_file() start() (IPython.parallel.apps.launcher.IPClusterLauncher
(IPython.parallel.apps.ipcontrollerapp.IPClusterEngines method), 644
method), 620 start() (IPython.parallel.apps.launcher.LocalControllerLauncher
method), 658
stage_default_config_file() start() (IPython.parallel.apps.launcher.LocalEngineLauncher
(IPython.parallel.apps.ipengineapp.IPEngineApp method), 661
method), 626 start() (IPython.parallel.apps.launcher.LocalEngineSetLauncher
method), 663
stage_default_config_file() start() (IPython.parallel.apps.launcher.LocalProcessLauncher
(IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 666
method), 634 start() (IPython.parallel.apps.launcher.LSFControllerLauncher
start() (in module IPython.utils.growl), 864 start() (IPython.parallel.apps.launcher.LSFEngineSetLauncher
method), 266 start() (IPython.parallel.apps.launcher.LSFLauncher
method), 648
start() (IPython.core.application.BaseIPythonApplication start() (IPython.parallel.apps.launcher.LSFControllerLauncher
method), 291 start() (IPython.parallel.apps.launcher.LSFEngineSetLauncher
method), 651
start() (IPython.core.history.HistorySavingThread start() (IPython.parallel.apps.launcher.LSFLauncher

method), 655
start() (IPython.parallel.apps.launcher.MPIExecController.start_data (IPython.parallel.engine.streamkernel.Kernel method), 669
start() (IPython.parallel.apps.launcher.MPIExecEngine.start_controller() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 672
start() (IPython.parallel.apps.launcher.MPIExecLauncher.start_data (IPython.parallel.apps.launcher.BaseLauncher method), 675
start() (IPython.parallel.apps.launcher.PBSController.start_data (IPython.parallel.apps.launcher.BatchSystemLauncher method), 678
start() (IPython.parallel.apps.launcher.PBSEngineSet.start_data (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 682
start() (IPython.parallel.apps.launcher.PBSLauncher.start_data (IPython.parallel.apps.launcher.LocalControllerLauncher method), 685
start() (IPython.parallel.apps.launcher.SGEController.start_data (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 689
start() (IPython.parallel.apps.launcher.SGEEngineSet.start_data (IPython.parallel.apps.launcher.LocalEngineSetLauncher method), 692
start() (IPython.parallel.apps.launcher.SGELauncher.start_data (IPython.parallel.apps.launcher.LocalProcessLauncher method), 696
start() (IPython.parallel.apps.launcher.SSHController.start_data (IPython.parallel.apps.launcher.LSFControllerLauncher method), 699
start() (IPython.parallel.apps.launcher.SSHEngineLauncher.start_data (IPython.parallel.apps.launcher.LSFEngineSetLauncher method), 702
start() (IPython.parallel.apps.launcher.SSHEngineSet.start_data (IPython.parallel.apps.launcher.LSFLauncher attribute), 705
start() (IPython.parallel.apps.launcher.SSHLauncher.start_data (IPython.parallel.apps.launcher.MPIExecControllerLauncher method), 708
start() (IPython.parallel.apps.launcher.WindowsHPCCController.start_data (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher method), 711
start() (IPython.parallel.apps.launcher.WindowsHPCEngine.start_data (IPython.parallel.apps.launcher.MPIExecLauncher method), 714
start() (IPython.parallel.apps.launcher.WindowsHPCCLauncher.start_data (IPython.parallel.apps.launcher.PBSControllerLauncher method), 717
start() (IPython.parallel.apps.logwatcher.LogWatcher.start_data (IPython.parallel.apps.launcher.PBSEngineSetLauncher method), 719
start() (IPython.parallel.apps.win32support.ForwarderThread.start_data (IPython.parallel.apps.launcher.PBSLauncher method), 721
start() (IPython.parallel.controller.heartmonitor.Heart.start_data (IPython.parallel.apps.launcher.SGEControllerLauncher method), 781
start() (IPython.parallel.controller.heartmonitor.HeartMonitor.start_data (IPython.parallel.apps.launcher.SGEEngineSetLauncher method), 783
start() (IPython.parallel.controller.hub.HubFactory.start_data (IPython.parallel.apps.launcher.SGELauncher method), 794
start() (IPython.parallel.controller.scheduler.TaskScheduler.start_data (IPython.parallel.apps.launcher.SSHControllerLauncher method), 798
start() (IPython.parallel.engine.EngineFactory.start_data (IPython.parallel.apps.launcher.SSHEngineLauncher method), 805
start() (IPython.parallel.engine.kernelstarter.KernelStart.start_data (IPython.parallel.apps.launcher.SSHEngineSetLauncher

attribute), 705
start_data (IPython.parallel.apps.launcher.SSHLauncher attribute), 708
start_data (IPython.parallel.apps.launcher.WindowsHPCController attribute), 711
start_data (IPython.parallel.apps.launcher.WindowsHPCEngine attribute), 714
start_data (IPython.parallel.apps.launcher.WindowsHPCLaunch attribute), 715
start_displayhook() (IPython.core.displayhook.DisplayHook method), 321
start_engines() (IPython.parallel.apps.ipclusterapp.IPClusterEng attribute), 550
start_engines() (IPython.parallel.apps.ipclusterapp.IPClusterStar attribute), 550
start_event_loop_qt4() (in IPython.lib.guisupport), 569
start_event_loop_wx() (in IPython.lib.guisupport), 569
start_ipython() (in IPython.testing.globalipapp), 834
start_kernel() (IPython.parallel.engine.kernelstarter.KernelStart attribute), 549
start_logging() (IPython.parallel.apps.ipclusterapp.IPClusterEng attribute), 550
start_logging() (IPython.parallel.apps.ipclusterapp.IPClusterStar attribute), 551
startswith (IPython.utils.text.LSString attribute), 894
startup_command (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 626
startup_script (IPython.parallel.apps.ipengineapp.IPEngineApp_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 551
stat_completed (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 548
stat_completed (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_completed (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 550
stat_completed_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_completed_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_completed_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 550
stat_created (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_created (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 549
stat_created_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 550
stat_created_c (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 550
stat_created_c (IPython.lib.backgroundjobs.BackgroundJobFunc attribute), 550
stat_dead (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_dead (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 550
stat_dead_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 550
stat_dead_c (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 550
stat_dead_c (IPython.lib.backgroundjobs.BackgroundJobFunc attribute), 551
stat_running (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 549
stat_running (IPython.lib.backgroundjobs.BackgroundJobExpr attribute), 550
stat_running (IPython.lib.backgroundjobs.BackgroundJobFunc attribute), 551
stat_running_c (IPython.lib.backgroundjobs.BackgroundJobBase attribute), 552
stb2text (IPython.core.ultratb.AutoFormattedTB method), 538
stb2text_c (IPython.core.ultratb.ColorTB method), 540
stb2text_c (IPython.core.ultratb.FormattedTB method), 541
stb2text_c (IPython.core.ultratb.ListTB method), 543
stb2text_c (IPython.core.ultratb.SyntaxTB method), 544
stb2text_c (IPython.core.ultratb.TBTools method), 545
stb2text_c (IPython.core.ultratb.VerboseTB method), 546
std_err_file_path (IPython.parallel.apps.winhpcjob.IPCControllerTask attribute), 549

attribute), 726
std_err_file_path (IPython.parallel.apps.winhpcjob.IPEngineLauncher.
attribute), 732
std_err_file_path (IPython.parallel.apps.winhpcjob.WindowsHPCControllerLauncher.
attribute), 738
std_out_file_path (IPython.parallel.apps.winhpcjob.IPCControllerLauncher.
attribute), 726
std_out_file_path (IPython.parallel.apps.winhpcjob.IPEngineLauncher.
attribute), 732
std_out_file_path (IPython.parallel.apps.winhpcjob.WindowsHPCControllerLauncher.
attribute), 738
stop() (IPython.core.history.HistorySavingThread
method), 364
stop() (IPython.parallel.apps.launcher.BaseLauncher
method), 638
stop() (IPython.parallel.apps.launcher.BatchSystemLauncher
method), 642
stop() (IPython.parallel.apps.launcher.IPClusterLauncher
method), 645
stop() (IPython.parallel.apps.launcher.LocalController
method), 658
stop() (IPython.parallel.apps.launcher.LocalEngineLauncher.
attribute), 661
stop() (IPython.parallel.apps.launcher.LocalEngineSetLauncher.
method), 663
stop() (IPython.parallel.apps.launcher.LocalProcessLauncher.
method), 666
stop() (IPython.parallel.apps.launcher.LSFControllerLauncher.
method), 648
stop() (IPython.parallel.apps.launcher.LSFEngineSetLauncher.
method), 651
stop() (IPython.parallel.apps.launcher.LSFLauncher
method), 655
stop() (IPython.parallel.apps.launcher.MPIExecController
method), 669
stop() (IPython.parallel.apps.launcher.MPIExecEngineController
method), 672
stop() (IPython.parallel.apps.launcher.MPIExecLauncher
method), 675
stop() (IPython.parallel.apps.launcher.PBSControllerLauncher.
method), 678
stop() (IPython.parallel.apps.launcher.PBSEngineSetLauncher.
method), 682
stop() (IPython.parallel.apps.launcher.PBSLauncher
method), 685
stop() (IPython.parallel.apps.launcher.SGEControllerLauncher.
method), 689
stop() (IPython.parallel.apps.launcher.SGEEngineSetLauncher.
method), 693
stop() (IPython.parallel.apps.launcher.SGEControllerLauncher.
method), 696
stop() (IPython.parallel.apps.launcher.SSHControllerLauncher.
method), 699
stop() (IPython.parallel.apps.launcher.SSHEngineLauncher.
method), 702
stop() (IPython.parallel.apps.launcher.SSHEngineSetLauncher.
method), 705
stop() (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher.
method), 711
stop() (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher.
method), 714
stop() (IPython.parallel.apps.logwatcher.LogWatcher
method), 719

attribute), 678
stop_data (IPython.parallel.apps.launcher.PBSEngineStream (IPython.testing.globalipapp.StreamProxy attribute), 682
stop_data (IPython.parallel.apps.launcher.PBSLaunchersStreamProxy (class in IPython.testing.globalipapp), attribute), 685
stop_data (IPython.parallel.apps.launcher.SGEControllerStream (in module IPython.utils.pickleshare), 882
attribute), 689
stop_data (IPython.parallel.apps.launcher.SGEEngineStreams in IPython.utils.ipstruct), 867
attribute), 693
stop_data (IPython.parallel.apps.launcher.SGELauncher (IPython.core.ultratb.AutoFormattedTB
attribute), 696
method), 538
stop_data (IPython.parallel.apps.launcher.SSHControllerStream (in module IPython.core.ultratb.traceback),
attribute), 699
stop_data (IPython.parallel.apps.launcher.SSHEngineLauncher 540
attribute), 702
structured_traceback()
stop_data (IPython.parallel.apps.launcher.SSHEngineSetLaunch (IPython.core.ultratb.FormattedTB
attribute), 705
method), 541
stop_data (IPython.parallel.apps.launcher.SSHLaunchersstructured_traceback() (IPython.core.ultratb.ListTB
attribute), 708
method), 543
stop_data (IPython.parallel.apps.launcher.WindowsHPCControllerStream (in module IPython.core.ultratb.SyntaxTB
attribute), 711
method), 545
stop_data (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher
attribute), 714
structured_traceback()
stop_data (IPython.parallel.apps.launcher.WindowsHPCLaunchers (IPython.core.ultratb.TBTools
attribute), 717
method), 545
stop_engines() (IPython.parallel.apps.ipclusterapp.IPClusterEnginesstructured_traceback()
method), 602
stop_engines() (IPython.parallel.apps.ipclusterapp.IPClusterStar 547
method), 608
subapp (IPython.config.application.Application attribute), 266
stop_here() (IPython.core.debugger.Pdb method),
313
subapp (IPython.core.application.BaseIPythonApplication
method), 602
stop_launchers() (IPython.parallel.apps.ipclusterapp.IPClusterEnginesattribute), 291
subapp (IPython.core.profileapp.ProfileApp attribute), 516
stop_launchers() (IPython.parallel.apps.ipclusterapp.IPClusterStarattribute), 516
subapp (IPython.core.profileapp.ProfileCreate attribute), 520
stop_now (IPython.core.history.HistorySavingThread
attribute), 364
subapp (IPython.core.profileapp.ProfileList attribute), 524
stop_receiving() (IPython.parallel.controller.scheduler.TaskScheduler), 524
subapp (IPython.parallel.apps.baseapp.BaseParallelApplication
attribute), 592
StopLocalExecution (class in IPython.parallel.error),
821
subapp (IPython.parallel.apps.ipclusterapp.IPClusterApp
attribute), 597
store_inputs() (IPython.core.history.HistoryManager
method), 362
subapp (IPython.parallel.apps.ipclusterapp.IPClusterEngines
attribute), 602
store_output() (IPython.core.history.HistoryManager
method), 363
subapp (IPython.parallel.apps.ipclusterapp.IPClusterStart
attribute), 608
str_safe() (in module IPython.core.prompts), 532
subapp (IPython.parallel.apps.ipclusterapp.IPClusterStop
attribute), 614
StrDispatch (class in IPython.utils.strdispatch), 885
stream (IPython.parallel.apps.logwatcher.LogWatcher

subapp (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 291
 attribute), 620
subapp (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 516
 attribute), 626
subapp (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 520
 attribute), 634
subcommand_description
 (IPython.config.application.Application attribute), 267
subcommand_description
 (IPython.core.application.BaseIPythonApplication attribute), 291
subcommand_description
 (IPython.core.profileapp.ProfileApp attribute), 516
subcommand_description
 (IPython.core.profileapp.ProfileCreate attribute), 520
subcommand_description
 (IPython.core.profileapp.ProfileList attribute), 524
subcommand_description
 (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 592
subcommand_description
 (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 597
subcommand_description
 (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 602
subcommand_description
 (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 608
subcommand_description
 (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 614
subcommand_description
 (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 620
subcommand_description
 (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 626
subcommand_description
 (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 635
subcommand_description
 (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 642
subcommand_description
 (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 648
subcommand_description
 (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 651
subcommand_description
 (IPython.parallel.apps.launcher.LSFLauncher attribute), 655
subcommand_description
 (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 679
subcommand_description
 (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 682
subcommand_description
 (IPython.parallel.apps.launcher.PBSLauncher attribute), 685
subcommand_description
 (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 689
subcommand_description
 (IPython.parallel.apps.launcher.SGEEngineSetLauncher attribute), 693
subcommand_description
 (IPython.parallel.apps.logwatcher.LogWatcher method), 719
subcommands (IPython.config.application.Application subscribe() method), 267
subcommands (IPython.core.application.BaseIPythonApplication attribute), 516
subcommands (IPython.parallel.controller.Dependency

attribute), 774
successful() (AsyncResult method), 169
successful() (IPython.parallel.client.asyncresult.AsyncResult method), 740
successful() (IPython.parallel.client.asyncresult.AsyncResult method), 742
successful() (IPython.parallel.client.asyncresult.AsyncResult method), 743
SVGFormatter (class in IPython.core.formatters), 354
swapcase (IPython.utils.text.LSString attribute), 894
switch_log() (IPython.core.logger.Logger method), 418
symmetric_difference
 (IPython.parallel.controller.dependency.Dependency attribute), 774
symmetric_difference_update
 (IPython.parallel.controller.dependency.Dependency attribute), 774
sync_imports() (IPython.parallel.client.view.DirectView method), 761
sync_results()
 (in IPython.parallel.client.view), 771
synchronize_with_editor()
 (in IPython.core.hooks), 368
SyntaxTB (class in IPython.core.ultratb), 543
sys_info() (in module IPython.utils.sysinfo), 887
system() (IPython.core.interactiveshell.InteractiveShell method), 414
system_piped() (IPython.core.interactiveshell.InteractiveShell method), 414
T
table (IPython.parallel.controller.sqlitedb.SQLiteDB attribute), 802
target_outdated() (in module IPython.utils.path), 879
target_update() (in module IPython.utils.path), 879
targets (IPython.parallel.client.view.DirectView attribute), 761
targets (IPython.parallel.client.view.LoadBalancedView attribute), 766
targets (IPython.parallel.client.view.View attribute), 770
targets (IPython.parallel.controller.scheduler.TaskScheduler attribute), 799
task (IPython.parallel.controller.hub.HubFactory attribute), 794
HubRef (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 727
MapRef (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 732
ResultId (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 738
task_name (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 727
task_name (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 732
task_name (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 738
TaskAborted (class in IPython.parallel.error), 821
TaskRejectError (class in IPython.parallel.error), 821
tasks (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 724
tasks (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 730
tasks (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 735
tasks (IPython.parallel.controller.hub.Hub attribute), 790
TaskScheduler (class in IPython.parallel.controller.scheduler), 795
TaskTimeout (class in IPython.parallel.error), 822
ShellOffset (IPython.core.ultratb.AutoFormattedTB attribute), 538
tb_offset (IPython.core.ultratb.ColorTB attribute), 540
tb_offset (IPython.core.ultratb.FormattedTB attribute), 542
tb_offset (IPython.core.ultratb.ListTB attribute), 543
tb_offset (IPython.core.ultratb.SyntaxTB attribute), 544
tb_offset (IPython.core.ultratb.TBTools attribute), 545
tb_offset (IPython.core.ultratb.VerboseTB attribute), 547
TBTools (class in IPython.core.ultratb), 545
TCPAddress (class in IPython.utils.traitslets), 927
tearDown() (IPython.testing.tools.TempFileMixin method), 846

Tee (class in IPython.utils.io), 865
temp_flags() (IPython.parallel.client.view.DirectView method), 761
temp_flags() (IPython.parallel.client.view.LoadBalancedView method), 766
temp_flags() (IPython.parallel.client.view.View method), 770
temp_pyfile() (in module IPython.utils.io), 867
TempFileMixin (class in IPython.testing.tools), 846
term_clear() (in module IPython.utils.terminal), 888
TermColors (class in IPython.utils.coloransi), 857
test() (in module IPython.testing), 829
test() (in module IPython.utils.pickleshare), 882
test_for() (in module IPython.testing.iptest), 836
test_trivial() (in module IPython.testing.plugin.test_refs), 845
Text (class in IPython.lib.pretty), 584
text() (IPython.core.ultratb.AutoFormattedTB method), 538
text() (IPython.core.ultratb.ColorTB method), 540
text() (IPython.core.ultratb.FormattedTB method), 542
text() (IPython.core.ultratb.ListTB method), 543
text() (IPython.core.ultratb.SyntaxTB method), 545
text() (IPython.core.ultratb.TBTools method), 545
text() (IPython.core.ultratb.VerboseTB method), 547
text() (IPython.lib.pretty.PrettyPrinter method), 583
text() (IPython.lib.pretty.RepresentationPrinter method), 584
This (class in IPython.utils.traits), 928
tic (IPython.parallel.controller.heartmonitor.HeartMonitor attribute), 783
timeout (IPython.parallel.client.view.LoadBalancedView attribute), 766
timeout (IPython.parallel.engine.EngineFactoryTracer attribute), 805
TimeoutError (class in IPython.parallel.error), 822
timing() (in module IPython.utils.timing), 899
timings() (in module IPython.utils.timing), 899
timings_out() (in module IPython.utils.timing), 899
title (IPython.utils.text.LSString attribute), 894
tkinter_clipboard_get() (in module IPython.lib.clipboard), 553
to_work_dir() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 592
to_work_dir() (IPython.parallel.apps.ipclusterapp.IPClusterEngine method), 602
to_work_dir() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 608
to_work_dir() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 614
to_work_dir() (IPython.parallel.apps.ipcontrollerapp.IPClusterControllerApp method), 620
to_work_dir() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 627
to_work_dir() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 635
toggle_set_term_title() (in module IPython.utils.terminal), 888
topics (IPython.parallel.apps.logwatcher.LogWatcher attribute), 719
tostring() (IPython.parallel.apps.winhpcjob.IPClusterJob method), 724
tostring() (IPython.parallel.apps.winhpcjob.IPEngineSetJob method), 730
tostring() (IPython.parallel.apps.winhpcjob.WinHPCJob method), 735
trace_dispatch() (IPython.core.debugger.Pdb method), 313
traceback (IPython.parallel.error.CompositeError attribute), 813
traceback (IPython.parallel.error.RemoteError attribute), 820
traceback() (IPython.lib.backgroundjobs.BackgroundJobBase method), 549
traceback() (IPython.lib.backgroundjobs.BackgroundJobExpr method), 550
traceback() (IPython.lib.backgroundjobs.BackgroundJobFunc method), 551
traceback() (IPython.lib.backgroundjobs.BackgroundJobManager method), 552
traceback() (IPython.lib.backgroundjobs.BackgroundJobManager method), 552
track (IPython.parallel.client.view.DirectView attribute), 761
track (IPython.parallel.client.view.LoadBalancedView attribute), 766
track (IPython.parallel.client.view.View attribute), 771
trait_metadata() (IPython.config.application.Application method), 267
trait_Application() (IPython.config.configurable.Configurable method), 270
trait_Engine() (IPython.config.configurable.LoggingConfigurable method), 272
trait_metadata() (IPython.config.configurable.SingletonConfigurable

method), 275
trait_metadata() (IPython.core.alias.AliasManager method), 286
trait_metadata() (IPython.core.application.BaseIPythonApplication trait_metadata() (IPython.core.prefilter.AssignmentChecker method), 470
method), 291
trait_metadata() (IPython.core.builtin_trap.BuiltinTrap trait_metadata() (IPython.core.prefilter.AssignSystemTransformer method), 468
method), 295
trait_metadata() (IPython.core.display_trap.DisplayTrap trait_metadata() (IPython.core.prefilter.AutoCallChecker method), 476
method), 318
trait_metadata() (IPython.core.displayhook.DisplayHook trait_metadata() (IPython.core.prefilter.AutoMagicChecker method), 474
method), 321
trait_metadata() (IPython.core.displaypub.DisplayPublisher trait_metadata() (IPython.core.prefilter.EmacsChecker method), 478
method), 324
trait_metadata() (IPython.core.extensions.ExtensionManager trait_metadata() (IPython.core.prefilter.EmacsHandler method), 479
method), 330
trait_metadata() (IPython.core.formatters.BaseFormatter trait_metadata() (IPython.core.prefilter.EscCharsChecker method), 481
method), 333
trait_metadata() (IPython.core.formatters.DisplayFormatter trait_metadata() (IPython.core.prefilter.HelpHandler method), 483
method), 336
trait_metadata() (IPython.core.formatters.HTMLFormatter trait_metadata() (IPython.core.prefilter.IPyAutocallChecker method), 485
method), 339
trait_metadata() (IPython.core.formatters.JavascriptFormatter trait_metadata() (IPython.core.prefilter.IPyPromptTransformer method), 487
method), 345
trait_metadata() (IPython.core.formatters.JSONFormatter trait_metadata() (IPython.core.prefilter.MacroChecker method), 489
method), 342
trait_metadata() (IPython.core.formatters.LatexFormatter trait_metadata() (IPython.core.prefilter.MacroHandler method), 491
method), 348
trait_metadata() (IPython.core.formatters.PlainTextFormatter trait_metadata() (IPython.core.prefilter.MagicHandler method), 493
method), 354
trait_metadata() (IPython.core.formatters.PNGFormatter trait_metadata() (IPython.core.prefilter.MultiLineMagicChecker method), 495
method), 351
trait_metadata() (IPython.core.formatters.SVGFormatter trait_metadata() (IPython.core.prefilter.PrefilterChecker method), 497
method), 357
trait_metadata() (IPython.core.history.HistoryManager trait_metadata() (IPython.core.prefilter.PrefilterHandler method), 499
method), 363
trait_metadata() (IPython.core.interactiveshell.InteractiveShell trait_metadata() (IPython.core.prefilter.PrefilterManager method), 502
method), 414
trait_metadata() (IPython.core.payload.PayloadManager trait_metadata() (IPython.core.prefilter.PrefilterTransformer method), 504
method), 455
trait_metadata() (IPython.core.plugin.Plugin trait_metadata() (IPython.core.prefilter.PyPromptTransformer method), 506
method), 458
trait_metadata() (IPython.core.plugin.PluginManager trait_metadata() (IPython.core.prefilter.PythonOpsChecker method), 508
method), 460
trait_metadata() (IPython.core.prefilter.AliasChecker trait_metadata() (IPython.core.prefilter.ShellEscapeChecker method), 510
method), 463
trait_metadata() (IPython.core.prefilter.AliasHandler trait_metadata() (IPython.core.prefilter.ShellEscapeHandler method), 512
method), 465
trait_metadata() (IPython.core.prefilter.AssignMagicTransformer trait_metadata() (IPython.core.profileapp.ProfileApp

method), 516
trait_metadata() (IPython.core.profileapp.ProfileCreate trait_metadata() (IPython.parallel.apps.launcher.MPIExecEngineSetL
method), 521 method), 672
trait_metadata() (IPython.core.profileapp.ProfileList trait_metadata() (IPython.parallel.apps.launcher.MPIExecLauncher
method), 524 method), 675
trait_metadata() (IPython.core.profiledir.ProfileDir trait_metadata() (IPython.parallel.apps.launcher.PBSControllerLaunc
method), 528 method), 679
trait_metadata() (IPython.core.shellapp.InteractiveShellApp metadata() (IPython.parallel.apps.launcher.PBSEngineSetLaunc
method), 535 method), 682
trait_metadata() (IPython.parallel.apps.baseapp.BaseParallelApp trait_metadata() (IPython.parallel.apps.launcher.PBSLauncher
method), 592 method), 686
trait_metadata() (IPython.parallel.apps.ipclusterapp.IPClusterApp trait_metadata() (IPython.parallel.apps.launcher.SGEControllerLaunc
method), 597 method), 689
trait_metadata() (IPython.parallel.apps.ipclusterapp.IPClusterEngines trait_metadata() (IPython.parallel.apps.launcher.SGEEngineSetLaunc
method), 602 method), 693
trait_metadata() (IPython.parallel.apps.ipclusterapp.IPClusterEngines trait_metadata() (IPython.parallel.apps.launcher.SGELauncher
method), 608 method), 696
trait_metadata() (IPython.parallel.apps.ipclusterapp.IPClusterEngines trait_metadata() (IPython.parallel.apps.launcher.SSHControllerLaunc
method), 614 method), 699
trait_metadata() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp trait_metadata() (IPython.parallel.apps.launcher.SSHEngineLauncher
method), 620 method), 702
trait_metadata() (IPython.parallel.apps.ipengineapp.IPEngineApp trait_metadata() (IPython.parallel.apps.launcher.SSHEngineSetLaunc
method), 627 method), 705
trait_metadata() (IPython.parallel.apps.ipengineapp.MPITrait_metadata() (IPython.parallel.apps.launcher.SSHLauncher
method), 629 method), 708
trait_metadata() (IPython.parallel.apps.iploggerapp.IPLingerApp trait_metadata() (IPython.parallel.apps.launcher.WindowsHPCContro
method), 635 method), 711
trait_metadata() (IPython.parallel.apps.launcher.BaseLauncher trait_metadata() (IPython.parallel.apps.launcher.WindowsHPCEngine
method), 638 method), 714
trait_metadata() (IPython.parallel.apps.launcher.BatchSystem trait_metadata() (IPython.parallel.apps.launcher.WindowsHPCLaunch
method), 642 method), 717
trait_metadata() (IPython.parallel.apps.launcher.IPCluster trait_metadata() (IPython.parallel.apps.logwatcher.LogWatcher
method), 645 method), 719
trait_metadata() (IPython.parallel.apps.launcher.LocalController trait_metadata() (IPython.parallel.apps.winhpcjob.IPCControllerJob
method), 658 method), 724
trait_metadata() (IPython.parallel.apps.launcher.LocalEngine trait_metadata() (IPython.parallel.apps.winhpcjob.IPCControllerTask
method), 661 method), 727
trait_metadata() (IPython.parallel.apps.launcher.LocalEngineSet trait_metadata() (IPython.parallel.apps.winhpcjob.IPEngineSetJob
method), 664 method), 730
trait_metadata() (IPython.parallel.apps.launcher.LocalProcess trait_metadata() (IPython.parallel.apps.winhpcjob.IPEngineTask
method), 666 method), 732
trait_metadata() (IPython.parallel.apps.launcher.LSFController trait_metadata() (IPython.parallel.apps.winhpcjob.WinHPCJob
method), 648 method), 735
trait_metadata() (IPython.parallel.apps.launcher.LSFEngine trait_metadata() (IPython.parallel.apps.winhpcjob.WinHPCTask
method), 652 method), 738
trait_metadata() (IPython.parallel.apps.launcher.LSFLauncher trait_metadata() (IPython.parallel.client.client.Client
method), 655 method), 750
trait_metadata() (IPython.parallel.apps.launcher.MPIExecutive trait_metadata() (IPython.parallel.client.view.DirectView

method), 761
trait_metadata() (IPython.parallel.client.view.LoadBalancedNames() (IPython.core.extensions.ExtensionManager method), 766
trait_metadata() (IPython.parallel.client.view.View trait_names() (IPython.core.formatters.BaseFormatter method), 771
trait_names() (IPython.core.formatters.DisplayFormatter method), 333
trait_metadata() (IPython.parallel.controller.dictdb.DictDB_names() (IPython.core.formatters.HTMLFormatter method), 777
trait_names() (IPython.core.formatters.HTMLFormatter method), 336
trait_metadata() (IPython.parallel.controller.dictdb.DictDB_names() (IPython.core.formatters.JavascriptFormatter method), 780
trait_names() (IPython.core.formatters.JavascriptFormatter method), 339
trait_metadata() (IPython.parallel.controller.heartmonitor.HeartMonitor trait_names() (IPython.core.formatters.JSONFormatter method), 784
trait_names() (IPython.core.formatters.JSONFormatter method), 345
trait_metadata() (IPython.parallel.controller.hub.EngineFactories() (IPython.core.formatters.LatexFormatter method), 785
trait_names() (IPython.core.formatters.LatexFormatter method), 342
trait_metadata() (IPython.parallel.controller.hub.HubFactoryNames() (IPython.core.formatters.PlainTextFormatter method), 790
trait_names() (IPython.core.formatters.PlainTextFormatter method), 348
trait_metadata() (IPython.parallel.controller.hub.HubFactories() (IPython.core.formatters.PNGFormatter method), 794
trait_names() (IPython.core.formatters.PNGFormatter method), 351
trait_metadata() (IPython.parallel.controller.scheduler.TaskScheduler trait_names() (IPython.core.formatters.SVGFormatter method), 799
trait_names() (IPython.core.formatters.SVGFormatter method), 357
trait_metadata() (IPython.parallel.controller.sqlitedb.SQLiteDatabases() (IPython.core.formatters.PLAINFORMATTER method), 802
trait_names() (IPython.core.formatters.PLAINFORMATTER method), 354
trait_metadata() (IPython.parallel.engine.EngineFactories() (IPython.core.history.HistoryManager method), 805
trait_names() (IPython.core.history.HistoryManager method), 363
trait_metadata() (IPython.parallel.engine.streamkernel.KernelNames() (IPython.core.interactiveshell.InteractiveShell method), 811
trait_names() (IPython.core.interactiveshell.InteractiveShell method), 414
trait_metadata() (IPython.parallel.factory.RegistrationFactoryNames() (IPython.core.payload.PayloadManager method), 825
trait_names() (IPython.core.payload.PayloadManager method), 455
trait_metadata() (IPython.utils.traits.HasTraits trait_names() (IPython.core.plugin.Plugin method), 920
trait_names() (IPython.core.plugin.Plugin method), 458
trait_names() (IPython.config.application.Application trait_names() (IPython.core.plugin.PluginManager method), 267
trait_names() (IPython.core.plugin.PluginManager method), 460
trait_names() (IPython.config.configurable.Configurable trait_names() (IPython.core.prefilter.AliasChecker method), 270
trait_names() (IPython.core.prefilter.AliasChecker method), 463
trait_names() (IPython.config.configurable.LoggingConfigurables() (IPython.core.prefilter.AliasHandler method), 272
trait_names() (IPython.core.prefilter.AliasHandler method), 465
trait_names() (IPython.config.configurable.SingletonConfigurables() (IPython.core.prefilter.AssignMagicTransformer method), 275
trait_names() (IPython.core.prefilter.AssignMagicTransformer method), 467
trait_names() (IPython.core.alias.AliasManager trait_names() (IPython.core.prefilter.AssignmentChecker method), 286
trait_names() (IPython.core.prefilter.AssignmentChecker method), 470
trait_names() (IPython.core.application.BaseIPythonApplication trait_names() (IPython.core.prefilter.AssignSystemTransformer method), 292
trait_names() (IPython.core.prefilter.AssignSystemTransformer method), 469
trait_names() (IPython.core.builtin_trap.BuiltinTrap trait_names() (IPython.core.prefilter.AutocallChecker method), 295
trait_names() (IPython.core.prefilter.AutocallChecker method), 476
trait_names() (IPython.core.display_trap.DisplayTrap trait_names() (IPython.core.prefilter.AutoHandler method), 318
trait_names() (IPython.core.prefilter.AutoHandler method), 472
trait_names() (IPython.core.displayhook.DisplayHook trait_names() (IPython.core.prefilter.AutoMagicChecker method), 321
trait_names() (IPython.core.prefilter.AutoMagicChecker method), 474
trait_names() (IPython.core.displaypub.DisplayPublisher trait_names() (IPython.core.prefilter.EmacsChecker

method), 478
trait_names() (IPython.core.prefilter.EmacsHandler trait_names() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 479 method), 602
trait_names() (IPython.core.prefilter.EscCharsChecker trait_names() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 481 method), 608
trait_names() (IPython.core.prefilter.HelpHandler trait_names() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 483 method), 614
trait_names() (IPython.core.prefilter.IPyAutocallCheck trait_names() (IPython.parallel.apps.ipcontrollerapp.IPControllerApp method), 485 method), 620
trait_names() (IPython.core.prefilter.IPyPromptTransform trait_names() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 487 method), 627
trait_names() (IPython.core.prefilter.MacroChecker trait_names() (IPython.parallel.apps.ipengineappMPI method), 489 method), 629
trait_names() (IPython.core.prefilter.MacroHandler trait_names() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 491 method), 635
trait_names() (IPython.core.prefilter.MagicHandler trait_names() (IPython.parallel.apps.launcher.BaseLauncher method), 493 method), 638
trait_names() (IPython.core.prefilter.MultiLineMagicCheck trait_names() (IPython.parallel.apps.launcher.BatchSystemLauncher method), 495 method), 642
trait_names() (IPython.core.prefilter.PrefilterChecker trait_names() (IPython.parallel.apps.launcher.IPClusterLauncher method), 497 method), 645
trait_names() (IPython.core.prefilter.PrefilterHandler trait_names() (IPython.parallel.apps.launcher.LocalControllerLauncher method), 499 method), 658
trait_names() (IPython.core.prefilter.PrefilterManager trait_names() (IPython.parallel.apps.launcher.LocalEngineLauncher method), 502 method), 661
trait_names() (IPython.core.prefilter.PrefilterTransform trait_names() (IPython.parallel.apps.launcher.LocalEngineSetLauncher method), 504 method), 664
trait_names() (IPython.core.prefilter.PyPromptTransform trait_names() (IPython.parallel.apps.launcher.LocalProcessLauncher method), 506 method), 666
trait_names() (IPython.core.prefilter.PythonOpsCheck trait_names() (IPython.parallel.apps.launcher.LSFControllerLauncher method), 508 method), 648
trait_names() (IPython.core.prefilter.ShellEscapeCheck trait_names() (IPython.parallel.apps.launcher.LSFEngineSetLauncher method), 510 method), 652
trait_names() (IPython.core.prefilter.ShellEscapeHandle trait_names() (IPython.parallel.apps.launcher.LSFLauncher method), 512 method), 655
trait_names() (IPython.core.profileapp.ProfileApp trait_names() (IPython.parallel.apps.launcher.MPIExecControllerLauncher method), 516 method), 669
trait_names() (IPython.core.profileapp.ProfileCreate trait_names() (IPython.parallel.apps.launcher.MPIExecEngineSetLauncher method), 521 method), 672
trait_names() (IPython.core.profileapp.ProfileList trait_names() (IPython.parallel.apps.launcher.MPIExecLauncher method), 524 method), 675
trait_names() (IPython.core.profiledir.ProfileDir trait_names() (IPython.parallel.apps.launcher.PBSControllerLauncher method), 528 method), 679
trait_names() (IPython.core.shellapp.InteractiveShellApp trait_names() (IPython.parallel.apps.launcher.PBSEngineSetLauncher method), 535 method), 682
trait_names() (IPython.parallel.apps.baseapp.BaseParallelApplication trait_names() (IPython.parallel.apps.launcher.PBSLauncher method), 592 method), 686
trait_names() (IPython.parallel.apps.ipclusterapp.IPCluster trait_names() (IPython.parallel.apps.launcher.SGEControllerLauncher

method), 689
trait_names() (IPython.parallel.apps.launcher.SGEEngine trait_names() (IPython.parallel.controller.hub.Hub method), 693
method), 786
trait_names() (IPython.parallel.apps.launcher.SGELauncher trait_names() (IPython.parallel.controller.hub.HubFactory method), 696
method), 791
trait_names() (IPython.parallel.apps.launcher.SSHController trait_names() (IPython.parallel.controller.hub.HubFactory method), 699
method), 794
trait_names() (IPython.parallel.apps.launcher.SSHEngine trait_names() (IPython.parallel.controller.sqlite.SQLiteDB method), 702
method), 802
trait_names() (IPython.parallel.apps.launcher.SSHEngine trait_names() (IPython.parallel.engine.EngineFactory method), 705
method), 805
trait_names() (IPython.parallel.apps.launcher.SSHLauncher trait_names() (IPython.parallel.engine.streamkernel.Kernel method), 708
method), 811
trait_names() (IPython.parallel.apps.launcher.WindowsHPCGames trait_names() (IPython.parallel.factory.RegistrationFactory method), 711
method), 825
trait_names() (IPython.parallel.apps.launcher.WindowsHPCEngines trait_names() (IPython.utils.traits.HasTraits method), 714
method), 920
trait_names() (IPython.parallel.apps.launcher.WindowsHPCMonitors trait_names() (IPython.utils.traits in IPython.utils.traits), 929
method), 717
traits() (IPython.config.application.Application method), 267
trait_names() (IPython.parallel.apps.logwatcher.LogWatcher traits() (IPython.config.configurable.Configurable method), 719
method), 267
traits() (IPython.config.configurable.LoggingConfigurable method), 724
method), 270
traits() (IPython.config.configurable.SingletonConfigurable method), 727
method), 272
traits() (IPython.core.alias.AliasManager method), 730
method), 275
trait_names() (IPython.parallel.apps.winhpcjob.IPControllerJob traits() (IPython.core.application.BaseIPythonApplication method), 276
method), 277
traits() (IPython.core.display_trap.DisplayTrap method), 286
method), 286
traits() (IPython.core.application.BaseIPythonApplication method), 292
method), 292
traits() (IPython.core.builtin_trap.BuiltinTrap method), 296
method), 296
traits() (IPython.core.display_trap.DisplayTrap method), 318
method), 318
traits() (IPython.core.displayhook.DisplayHook method), 321
method), 321
traits() (IPython.core.displaypub.DisplayPublisher method), 324
method), 324
traits() (IPython.core.extensions.ExtensionManager method), 330
method), 330
traits() (IPython.core.formatters.BaseFormatter method), 333
method), 333
traits() (IPython.core.formatters.DisplayFormatter method), 336
method), 336
traits() (IPython.core.formatters.HTMLFormatter method), 339
method), 339
traits() (IPython.core.formatters.JavascriptFormatter method), 345
method), 345

| | | | |
|----------|--|----------|---|
| traits() | (IPython.core.formatters.JSONFormatter method), 342 | traits() | (IPython.core.prefilter.MacroHandler method), 489 |
| traits() | (IPython.core.formatters.LatexFormatter method), 348 | traits() | (IPython.core.prefilter.MagicHandler method), 491 |
| traits() | (IPython.core.formatters.PlainTextFormatter method), 354 | traits() | (IPython.core.prefilter.MultiLineMagicChecker method), 493 |
| traits() | (IPython.core.formatters.PNGFormatter method), 351 | traits() | (IPython.core.prefilter.PrefilterChecker method), 495 |
| traits() | (IPython.core.formatters.SVGFormatter method), 357 | traits() | (IPython.core.prefilter.PrefilterHandler method), 497 |
| traits() | (IPython.core.history.HistoryManager method), 363 | traits() | (IPython.core.prefilter.PrefilterManager method), 499 |
| traits() | (IPython.core.interactiveshell.InteractiveShell method), 414 | traits() | (IPython.core.prefilter.PrefilterTransformer method), 502 |
| traits() | (IPython.core.payload.PayloadManager method), 455 | traits() | (IPython.core.prefilter.PyPromptTransformer method), 504 |
| traits() | (IPython.core.plugin.Plugin method), 458 | traits() | (IPython.core.prefilter.PythonOpsChecker method), 506 |
| traits() | (IPython.core.plugin.PluginManager method), 460 | traits() | (IPython.core.prefilter.ShellEscapeChecker method), 508 |
| traits() | (IPython.core.prefilter.AliasChecker method), 463 | traits() | (IPython.core.prefilter.ShellEscapeHandler method), 510 |
| traits() | (IPython.core.prefilter.AliasHandler method), 465 | traits() | (IPython.core.profileapp.ProfileApp method), 512 |
| traits() | (IPython.core.prefilter.AssignMagicTransformer method), 467 | traits() | (IPython.core.profileapp.ProfileCreate method), 516 |
| traits() | (IPython.core.prefilter.AssignmentChecker method), 470 | traits() | (IPython.core.profileapp.ProfileList method), 521 |
| traits() | (IPython.core.prefilter.AssignSystemTransformer method), 469 | traits() | (IPython.core.profiledir.ProfileDir method), 525 |
| traits() | (IPython.core.prefilter.AutocallChecker method), 476 | traits() | (IPython.core.shellapp.InteractiveShellApp method), 529 |
| traits() | (IPython.core.prefilter.AutoHandler method), 472 | traits() | (IPython.core.shellapp.InteractiveShellApp method), 535 |
| traits() | (IPython.core.prefilter.AutoMagicChecker method), 474 | traits() | (IPython.parallel.apps.baseapp.BaseParallelApplication method), 592 |
| traits() | (IPython.core.prefilter.EmacsChecker method), 478 | traits() | (IPython.parallel.apps.ipclusterapp.IPClusterApp method), 597 |
| traits() | (IPython.core.prefilter.EmacsHandler method), 480 | traits() | (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 603 |
| traits() | (IPython.core.prefilter.EscCharsChecker method), 481 | traits() | (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 608 |
| traits() | (IPython.core.prefilter.HelpHandler method), 483 | traits() | (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 614 |
| traits() | (IPython.core.prefilter.IPyAutocallChecker method), 485 | traits() | (IPython.parallel.apps.ipcontrollerapp.IPClusterStop method), 620 |
| traits() | (IPython.core.prefilter.IPyPromptTransformer method), 487 | traits() | (IPython.parallel.apps.ipengineapp.IPEngineApp method), 627 |
| traits() | (IPython.core.prefilter.MacroChecker method), 489 | traits() | (IPython.parallel.apps.ipengineapp.MPI method), 629 |

method), 629
traits() (IPython.parallel.apps.iploggerapp.IPLoggerApp traits() (IPython.parallel.apps.launcher.WindowsHPCControllerLauncher method), 711
method), 635
traits() (IPython.parallel.apps.launcher.BaseLauncher traits() (IPython.parallel.apps.launcher.WindowsHPCEngineSetLauncher method), 714
method), 638
traits() (IPython.parallel.apps.launcher.BatchSystemLauncher traits() (IPython.parallel.apps.launcher.WindowsHPCLauncher method), 717
method), 642
traits() (IPython.parallel.apps.launcher.IPClusterLauncher traits() (IPython.parallel.apps.logwatcher.LogWatcher method), 719
method), 645
traits() (IPython.parallel.apps.launcher.LocalController traits() (IPython.parallel.apps.winhpcjob.IPCControllerJob method), 724
method), 658
traits() (IPython.parallel.apps.launcher.LocalEngineLauncher traits() (IPython.parallel.apps.winhpcjob.IPCControllerTask method), 727
method), 661
traits() (IPython.parallel.apps.launcher.LocalEngineSet traits() (IPython.parallel.apps.winhpcjob.IPEngineSetJob method), 730
method), 664
traits() (IPython.parallel.apps.launcher.LocalProcessLauncher traits() (IPython.parallel.apps.winhpcjob.IPEngineTask method), 732
method), 666
traits() (IPython.parallel.apps.launcher.LSFController traits() (IPython.parallel.apps.winhpcjob.WinHPCJob method), 736
method), 648
traits() (IPython.parallel.apps.launcher.LSFEngineSet traits() (IPython.parallel.apps.winhpcjob.WinHPCTask method), 738
method), 652
traits() (IPython.parallel.apps.launcher.LSFLauncher traits() (IPython.parallel.client.client.Client method), 750
method), 655
traits() (IPython.parallel.apps.launcher.MPIExecController traits() (IPython.parallel.client.view.DirectView method), 762
method), 669
traits() (IPython.parallel.apps.launcher.MPIExecEngine traits() (IPython.parallel.client.view.LoadBalancedView method), 767
method), 672
traits() (IPython.parallel.apps.launcher.MPIExecLauncher traits() (IPython.parallel.client.view.View method), 771
method), 675
traits() (IPython.parallel.apps.launcher.PBSController traits() (IPython.parallel.controller.dictdb.BaseDB method), 778
method), 679
traits() (IPython.parallel.apps.launcher.PBSEngineSet traits() (IPython.parallel.controller.dictdb.DictDB method), 780
method), 682
traits() (IPython.parallel.apps.launcher.PBSLauncher traits() (IPython.parallel.controller.heartmonitor.HeartMonitor method), 784
method), 686
traits() (IPython.parallel.apps.launcher.SGEController traits() (IPython.parallel.controller.hub.EngineConnector method), 786
method), 689
traits() (IPython.parallel.apps.launcher.SGEEngineSet traits() (IPython.parallel.controller.hub.Hub method), 791
method), 693
traits() (IPython.parallel.apps.launcher.SGELauncher traits() (IPython.parallel.controller.hub.HubFactory method), 794
method), 696
traits() (IPython.parallel.apps.launcher.SSHController traits() (IPython.parallel.controller.scheduler.TaskScheduler method), 799
method), 699
traits() (IPython.parallel.apps.launcher.SSHEngineLauncher traits() (IPython.parallel.controller.sqlitedb.SQLiteDB method), 802
method), 702
traits() (IPython.parallel.apps.launcher.SSHEngineSet traits() (IPython.parallel.engine.engine.EngineFactory method), 805
method), 705
traits() (IPython.parallel.apps.launcher.SSHLauncher traits() (IPython.parallel.engine.streamkernel.Kernel

method), 811
 traits() (IPython.parallel.factory.RegistrationFactory
 method), 825
 traits() (IPython.utils.traits.HasTraits
 method), 920
 TraitType (class in IPython.utils.traits), 930
 transform() (IPython.core.prefilter.AssignMagicTransformer
 method), 467
 transform() (IPython.core.prefilter.AssignSystemTransformer
 method), 469
 transform() (IPython.core.prefilter.IPyPromptTransformer
 method), 487
 transform() (IPython.core.prefilter.PrefilterTransformer
 method), 504
 transform() (IPython.core.prefilter.PyPromptTransformer
 method), 506
 transform_alias() (IPython.core.alias.AliasManager
 method), 286
 transform_assign_magic() (in
 IPython.core.inputsplitter), 374
 transform_assign_system() (in
 IPython.core.inputsplitter), 374
 transform_classic_prompt() (in
 IPython.core.inputsplitter), 374
 transform_help_end() (in
 IPython.core.inputsplitter), 374
 transform_ipy_prompt() (in
 IPython.core.inputsplitter), 374
 transform_line() (IPython.core.prefilter.PrefilterManager
 method), 502
 transformers (IPython.core.prefilter.PrefilterManager
 attribute), 502
 translate (IPython.utils.text.LSString attribute), 895
 transport (IPython.parallel.controller.hub.HubFactory
 attribute), 794
 transport (IPython.parallel.engine.EngineFactory
 attribute), 805
 transport (IPython.parallel.factory.RegistrationFactory
 attribute), 825
 try_import() (in module IPython.core.completerlib),
 304
 TryNext (class in IPython.core.error), 326
 Tuple (class in IPython.utils.traits), 931
 twobin() (in
 IPython.parallel.controller.scheduler),
 800
 Type (class in IPython.utils.traits), 932
 type_printers (IPython.core.formatters.BaseFormatter
 attribute), 334
 type_printers (IPython.core.formatters.HTMLFormatter
 attribute), 339
 type_printers (IPython.core.formatters.JavascriptFormatter
 attribute), 345
 type_printers (IPython.core.formatters.JSONFormatter
 attribute), 342
 type_printers (IPython.core.formatters.LatexFormatter
 attribute), 348
 type_printers (IPython.core.formatters.PlainTextFormatter
 attribute), 354
 type_printers (IPython.core.formatters.PNGFormatter
 attribute), 351
 type_printers (IPython.core.formatters.SVGFormatter
 attribute), 357

U

unassigned (IPython.parallel.controller.hub.Hub at-
 tribute), 791
 uncache() (IPython.utils.pickleshare.PickleShareDB
 method), 881
 uncan() (in module IPython.utils.pickleutil), 883
 uncanDict() (in module IPython.utils.pickleutil), 883
 uncanSequence() (in
 IPython.utils.pickleutil), 883
 undefine_alias() (IPython.core.alias.AliasManager
 method), 286
 Undefined (in module IPython.utils.traits), 933
 endoc_header (IPython.core.debugger.Pdb at-
 tribute), 313
 Unicode (class in IPython.utils.traits), 933
 union (IPython.parallel.controller.dependency.Dependency
 attribute), 774
 uniq_stable() (in module IPython.utils.data), 859
 unit_type (IPython.parallel.apps.winhpcjob.IPCControllerJob
 attribute), 724
 unit_type (IPython.parallel.apps.winhpcjob.IPCControllerTask
 attribute), 727
 unit_type (IPython.parallel.apps.winhpcjob.IPEngineSetJob
 attribute), 730
 unit_type (IPython.parallel.apps.winhpcjob.IPEngineTask
 attribute), 733
 unit_type (IPython.parallel.apps.winhpcjob.WinHPCJob
 attribute), 736
 unit_type (IPython.parallel.apps.winhpcjob.WinHPCTask
 attribute), 739
 UnknownStatus (class
 in
 IPython.parallel.apps.launcher), 708

unload_extension() (IPython.core.extensions.ExtensionManager method), 330
UnmetDependency (class in IPython.parallel.error), 822
unpack_apply_message() (in module IPython.parallel.util), 829
UnpickleableException (class in IPython.parallel.error), 822
unquote_ends() (in module IPython.utils.text), 899
unreachable() (IPython.parallel.controller.dependency.Dependency method), 774
unregister_checker() (IPython.core.prefilter.PrefilterManager method), 502
unregister_engine() (IPython.parallel.controller.hub.Hub update_config() (IPython.core.profileapp.ProfileApp method), 516
method), 791
unregister_handler() (IPython.core.prefilter.PrefilterManager method), 502
unregister_plugin() (IPython.core.plugin.PluginManager method), 460
unregister_transformer() (IPython.core.prefilter.PrefilterManager method), 502
deserialize() (in module IPython.utils.newserialized), 875
deserialize_object() (in module IPython.parallel.util), 829
UnSerialized (class in IPython.utils.newserialized), 875
UnSerializeIt (class in IPython.utils.newserialized), 874
unset() (IPython.core.display_trap.DisplayTrap method), 318
unwrap_exception() (in module IPython.parallel.error), 823
update (IPython.config.loader.Config attribute), 279
update (IPython.parallel.client.client.Metadata attribute), 752
update (IPython.parallel.controller.dependency.Dependency attribute), 774
update (IPython.parallel.util.Namespace attribute), 827
update (IPython.parallel.util.ReverseDict attribute), 828
update (IPython.utils.coloransi.ColorSchemeTable attribute), 856
update (IPython.utils.ipstruct.Struct attribute), 871
Metadata (IPython.parallel.client.view.DirectView method), 762
update() (IPython.testing.globalipapp.ipnsdict method), 834
update() (IPython.utils.pickleshare.PickleShareDB method), 881
update_config() (IPython.config.application.Application method), 267
update_config() (IPython.core.application.BaseIPythonApplication method), 292
update_config() (IPython.core.profileapp.ProfileApp method), 516
update_config() (IPython.core.profileapp.ProfileCreate method), 521
update_config() (IPython.core.profileapp.ProfileList method), 525
update_config() (IPython.parallel.apps.baseapp.BaseParallelApplication method), 592
update_config() (IPython.parallel.apps.ipclusterapp.IPClusterApp method), 597
update_config() (IPython.parallel.apps.ipclusterapp.IPClusterEngines method), 603
update_config() (IPython.parallel.apps.ipclusterapp.IPClusterStart method), 609
update_config() (IPython.parallel.apps.ipclusterapp.IPClusterStop method), 614
update_config() (IPython.parallel.apps.ipcontrollerapp.IPControllerA method), 621
update_config() (IPython.parallel.apps.ipengineapp.IPEngineApp method), 627
update_config() (IPython.parallel.apps.iploggerapp.IPLLoggerApp method), 635
update_graph() (IPython.parallel.controller.scheduler.TaskScheduler method), 799
update_record() (IPython.parallel.controller.dictdb.DictDB method), 780
update_record() (IPython.parallel.controller.sqlitedb.SQLiteDatabase method), 802
update_user_ns() (IPython.core.displayhook.DisplayHook method), 321
upgrade_dir() (in module IPython.utils.upgradedir), 935
upper (IPython.utils.text.LSString attribute), 895
url (IPython.parallel.apps.logwatcher.LogWatcher attribute), 720
url (IPython.parallel.controller.hub.HubFactory attribute), 794
url (IPython.parallel.engine.engine.EngineFactory

V

| | |
|---|--|
| attribute), 806 | validate() (IPython.core.interactiveshell.SeparateUnicode method), 416 |
| url (IPython.parallel.factory.RegistrationFactory attribute), 825 | validate() (IPython.utils.traits.Bool method), 903 |
| url_file (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 627 | validate() (IPython.utils.traits.Bytes method), 904 |
| url_file_name (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 627 | validate() (IPython.utils.traits.CaselessStrEnum method), 911 |
| usage (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 603 | validate() (IPython.utils.traits.CBool method), 904 |
| usage (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 609 | validate() (IPython.utils.traits.CBytes method), 905 |
| UsageError (class in IPython.core.error), 327 | validate() (IPython.utils.traits.CComplex method), 906 |
| use (IPython.parallel.apps.ipengineapp.MPI attribute), 629 | validate() (IPython.utils.traits.CFloat method), 907 |
| use_rawinput (IPython.core.debugger.Pdb attribute), 313 | validate() (IPython.utils.traits.CInt method), 908 |
| use_threads (IPython.parallel.apps.ipcontrollerapp.IPCControllerApp attribute), 621 | validate() (IPython.utils.traits.CLong method), 909 |
| user (IPython.parallel.apps.launcher.SSHControllerLauncher attribute), 699 | validate() (IPython.utils.traits.Complex method), 913 |
| user (IPython.parallel.apps.launcher.SSHEngineLauncher attribute), 702 | validate() (IPython.utils.traits.Container method), 915 |
| user (IPython.parallel.apps.launcher.SSHLauncher attribute), 708 | validate() (IPython.utils.traits.CUnicode method), 910 |
| user_aliases (IPython.core.alias.AliasManager attribute), 286 | validate() (IPython.utils.traits.Dict method), 916 |
| user_call() (IPython.core.debugger.Pdb method), 313 | validate() (IPython.utils.traits.DottedObjectName method), 917 |
| user_exception() (IPython.core.debugger.Pdb method), 313 | validate() (IPython.utils.traits.Enum method), 918 |
| user_expressions() (IPython.core.interactiveshell.InteractiveShell method), 414 | validate() (IPython.utils.traits.Float method), 919 |
| user_line() (IPython.core.debugger.Pdb method), 313 | validate() (IPython.utils.traits.Instance method), 921 |
| user_ns (IPython.parallel.engine.EngineFactory attribute), 806 | validate() (IPython.utils.traits.Int method), 922 |
| user_ns (IPython.parallel.engine.streamkernel.Kernel attribute), 811 | validate() (IPython.utils.traits.List method), 923 |
| user_return() (IPython.core.debugger.Pdb method), 314 | validate() (IPython.utils.traits.Long method), 924 |
| user_variables() (IPython.core.interactiveshell.InteractiveShell method), 414 | validate() (IPython.utils.traits.ObjectName method), 926 |
| username (IPython.parallel.apps.winhpcjob.IPCControllerJob attribute), 724 | validate() (IPython.utils.traits.Set method), 927 |
| username (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 730 | validate() (IPython.utils.traits.TCPAddress method), 928 |
| username (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 736 | validate() (IPython.utils.traits.This method), 929 |
| | validate() (IPython.utils.traits.Tuple method), 932 |
| | validate() (IPython.utils.traits.Type method), 933 |
| | validate() (IPython.utils.traits.Unicode method), 934 |
| | validate_alias() (IPython.core.alias.AliasManager method), 286 |
| | validate_elements() (IPython.utils.traits.Container method), 286 |

method), 915
validate_elements() (IPython.utils.traitlets.List method), 923
validate_elements() (IPython.utils.traitlets.Set method), 927
validate_elements() (IPython.utils.traitlets.Tuple method), 932
validate_url() (in module IPython.parallel.util), 829
validate_url_container() (in module IPython.parallel.util), 829
values (IPython.config.loader.Config attribute), 280
values (IPython.parallel.client.client.Metadata attribute), 752
values (IPython.parallel.util.Namespace attribute), 827
values (IPython.parallel.util.ReverseDict attribute), 828
values (IPython.testing.globalipapp.ipnsdict attribute), 834
values (IPython.utils.coloransi.ColorSchemeTable attribute), 856
values (IPython.utils.ipstruct.Struct attribute), 871
values() (IPython.utils.pickleshare.PickleShareDB method), 881
var_expand() (IPython.core.interactiveshell.InteractiveShell method), 415
verbose (IPython.core.formatters.PlainTextFormatter attribute), 354
verbose() (IPython.core.ultratb.AutoFormattedTB method), 539
verbose() (IPython.core.ultratb.ColorTB method), 540
verbose() (IPython.core.ultratb.FormattedTB method), 542
VerboseTB (class in IPython.core.ultratb), 546
version (IPython.config.application.Application attribute), 267
version (IPython.core.application.BaseIPythonApplication attribute), 292
version (IPython.core.profileapp.ProfileApp attribute), 516
version (IPython.core.profileapp.ProfileCreate attribute), 521
version (IPython.core.profileapp.ProfileList attribute), 525
version (IPython.parallel.apps.baseapp.BaseParallelApp attribute), 592
version (IPython.parallel.apps.ipclusterapp.IPClusterApp attribute), 597
version (IPython.parallel.apps.ipclusterapp.IPClusterEngines attribute), 603
version (IPython.parallel.apps.ipclusterapp.IPClusterStart attribute), 609
version (IPython.parallel.apps.ipclusterapp.IPClusterStop attribute), 614
version (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 621
version (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 627
version (IPython.parallel.apps.iploggerapp.IPLoggerApp attribute), 635
version (IPython.parallel.apps.winhpcjob.IPControllerJob attribute), 724
version (IPython.parallel.apps.winhpcjob.IPControllerTask attribute), 727
version (IPython.parallel.apps.winhpcjob.IPEngineSetJob attribute), 730
version (IPython.parallel.apps.winhpcjob.IPEngineTask attribute), 733
version (IPython.parallel.apps.winhpcjob.WinHPCJob attribute), 736
version (IPython.parallel.apps.winhpcjob.WinHPCTask attribute), 739
vformat() (IPython.utils.text.EvalFormatter method), 890
View (class in IPython.parallel.client.view), 767
view (IPython.parallel.client.remotefunction.ParallelFunction attribute), 755
view (IPython.parallel.client.remotefunction.RemoteFunction attribute), 755

W

wait() (AsyncResult method), 169
wait() (IPython.parallel.client.asyncresult.AsyncHubResult method), 740
wait() (IPython.parallel.client.asyncresult.AsyncMapResult method), 742
wait() (IPython.parallel.client.asyncresult.AsyncResult method), 743
wait() (IPython.parallel.client.client.Client method), 751
wait() (IPython.parallel.client.view.DirectView method), 762
wait() (IPython.parallel.client.view.LoadBalancedView method), 767

wait() (IPython.parallel.client.view.View method), work_dir (IPython.parallel.apps.iploggerapp.IPLLoggerApp attribute), 635
771
wait_for_send() (IPython.parallel.client.asyncresult.AsyncResult method), 638
work_dir (IPython.parallel.apps.launcher.BaseLauncher attribute), 638
wait_for_send() (IPython.parallel.client.asyncresult.AsyncResult method), 642
work_dir (IPython.parallel.apps.launcher.BatchSystemLauncher attribute), 642
wait_for_send() (IPython.parallel.client.asyncresult.AsyncResult method), 645
work_dir (IPython.parallel.apps.launcher.IPClusterLauncher attribute), 645
wait_for_url_file (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 658
work_dir (IPython.parallel.apps.launcher.LocalControllerLauncher attribute), 658
waitget() (IPython.utils.pickleshare.PickleShareDB method), 661
work_dir (IPython.parallel.apps.launcher.LocalEngineLauncher attribute), 661
warn() (in module IPython.utils.warn), 936
work_dir (IPython.parallel.apps.launcher.LocalEngineSetLauncher attribute), 664
weighted() (in module IPython.parallel.controller.scheduler), 667
work_dir (IPython.parallel.apps.launcher.LocalProcessLauncher attribute), 667
White (IPython.utils.coloransi.InputTermColors attribute), 857
work_dir (IPython.parallel.apps.launcher.LSFControllerLauncher attribute), 648
White (IPython.utils.coloransi.TermColors attribute), 858
work_dir (IPython.parallel.apps.launcher.LSFEngineSetLauncher attribute), 652
wildcards_case_sensitive (IPython.core.interactiveshell.InteractiveShell attribute), 655
work_dir (IPython.parallel.apps.launcher.LSFLauncher attribute), 655
win32_clipboard_get() (in module IPython.lib.clipboard), 670
work_dir (IPython.parallel.apps.launcher.MPIExecControllerLauncher attribute), 670
WindowsHPCControllerLauncher (class in IPython.parallel.apps.launcher), 673
work_dir (IPython.parallel.apps.launcher.MPIExecLauncher attribute), 673
WindowsHPCEngineSetLauncher (class in IPython.parallel.apps.launcher), 675
work_dir (IPython.parallel.apps.launcher.PBSControllerLauncher attribute), 679
WindowsHPCLauncher (class in IPython.parallel.apps.launcher), 682
work_dir (IPython.parallel.apps.launcher.PBSEngineSetLauncher attribute), 682
WinHPCJob (class in IPython.parallel.apps.winhpcjob), 686
work_dir (IPython.parallel.apps.launcher.PBSLauncher attribute), 686
WinHPCTask (class in IPython.parallel.apps.winhpcjob), 689
work_dir (IPython.parallel.apps.launcher.SGEControllerLauncher attribute), 689
with_obj() (in module IPython.utils.attic), 851
work_dir (IPython.parallel.apps.baseapp.BaseParallelApplication attribute), 693
work_dir (IPython.parallel.apps.ipclusterapp.IPClusterEngineApp attribute), 696
work_dir (IPython.parallel.apps.ipclusterapp.IPClusterStark_dir attribute), 699
work_dir (IPython.parallel.apps.ipclusterapp.IPClusterStop_dir attribute), 702
work_dir (IPython.parallel.apps.ipcontrollerapp.IPControllerApp attribute), 705
work_dir (IPython.parallel.apps.ipengineapp.IPEngineApp attribute), 708
work_dir (IPython.parallel.apps.ipengineapp.IPEngineApp_dir attribute), 708

work_dir (IPython.parallel.apps.launcher.WindowsHPCController method), 652
attribute), 711 write_batch_script()
work_dir (IPython.parallel.apps.launcher.WindowsHPCEngineSet method), 655
attribute), 714 method), 655
work_dir (IPython.parallel.apps.launcher.WindowsHPCController method), 655
attribute), 717 (IPython.parallel.apps.launcher.PBSControllerLauncher
attribute), 727 write_batch_script()
work_directory (IPython.parallel.apps.winhpcjob.IPControllerTask method), 679
attribute), 733 method), 682
work_directory (IPython.parallel.apps.winhpcjob.WINHPCTask method), 679
attribute), 739 (IPython.parallel.apps.launcher.PBSLauncher
wrap_deprecated() (in module IPython.utils.attic), 686
851 method), 686
wrap_exception() (in module IPython.parallel.error), 690
823 method), 690
wrap_paragraphs() (in module IPython.utils.text), 693
899 write_batch_script()
write() (IPython.core.interactiveshell.InteractiveShell
method), 415 (IPython.parallel.apps.launcher.SGEControllerLauncher
method), 693
write() (IPython.core.prompts.BasePrompt method), 696
530 method), 696
write() (IPython.core.prompts.Prompt1 method), 415
531 write_err() (IPython.core.interactiveshell.InteractiveShell
method), 415
write() (IPython.core.prompts.Prompt2 method), 697
531 write_format_data()
write() (IPython.core.prompts.PromptOut method), 321 (IPython.core.displayhook.DisplayHook
method), 321
write() (IPython.parallel.apps.winhpcjob.IPControllerJob
method), 711 write_job_file() (IPython.parallel.apps.launcher.WindowsHPCController
method), 711
method), 724 write_job_file() (IPython.parallel.apps.launcher.WindowsHPCEngine
method), 714
write() (IPython.parallel.apps.winhpcjob.IPEngineSetJob
method), 730 write_job_file() (IPython.parallel.apps.launcher.WindowsHPCController
method), 717
method), 730
write() (IPython.parallel.apps.winhpcjob.WinHPCJob
method), 736 write_output_prompt()
method), 833 (IPython.core.displayhook.DisplayHook
method), 321
write() (IPython.testing.mkdoctests.IndentOut
method), 839 write_payload() (IPython.core.payload.PayloadManager
method), 455
write() (IPython.utils.io.IOStream method), 865 write_pid_file() (IPython.parallel.apps.baseapp.BaseParallelApplication
method), 592
write() (IPython.utils.io.Tee method), 866 write_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterEngines
method), 592
write_batch_script() write_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterStart
method), 603
method), 642 write_pid_file() (IPython.parallel.apps.ipclusterapp.IPClusterStop
method), 609
write_batch_script() write_pid_file() (IPython.parallel.apps.ipcontrollerapp.IPClusterStop
method), 614
method), 648
write_batch_script() write_pid_file() (IPython.parallel.apps.ipcontrollerapp.IPClusterA
method), 621

write_pid_file() (IPython.parallel.apps.ipengineapp.IPEngineApp
method), [627](#)
write_pid_file() (IPython.parallel.apps.iploggerapp.IPLLoggerApp
method), [635](#)
writelnes() (IPython.testing.globalipapp.StreamProxy
method), [833](#)
writelnes() (IPython.utils.io.IOStream method), [865](#)
writeout_cache() (IPython.core.history.HistoryManager
method), [363](#)

X

xmlns (IPython.parallel.apps.winhpcjob.IPControllerJob
attribute), [724](#)
xmlns (IPython.parallel.apps.winhpcjob.IPEngineSetJob
attribute), [730](#)
xmlns (IPython.parallel.apps.winhpcjob.WinHPCJob
attribute), [736](#)
xmode (IPython.core.interactiveshell.InteractiveShell
attribute), [415](#)
xsys() (in module IPython.testing.globalipapp), [834](#)

Y

Yellow (IPython.utils.coloransi.InputTermColors at-
tribute), [857](#)
Yellow (IPython.utils.coloransi.TermColors at-
tribute), [858](#)

Z

zfill (IPython.utils.text.LSString attribute), [895](#)
ZMQExitAutocall (class in IPython.core.autocall),
[293](#)