IPython Documentation

Release 1.2.1: An Afternoon Hack

The IPython Development Team

September 25, 2015

Contents

CHAPTER 1

Introduction

1.1 Overview

One of Python's most useful features is its interactive interpreter. It allows for 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 three main components:

- An enhanced interactive Python shell.
- A decoupled *two-process communication model*, which allows for multiple clients to connect to a computation kernel, most notably the web-based *notebook*
- 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 tab-completion, object introspection, system shell access, command history retrieval across sessions, 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. An interactive IPython shell 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.
3. Offer a flexible framework which can be used as the base environment for working with other systems, with Python as the underlying bridge 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, WX, GLUT, and OS X 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.
- 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 when you run the `%store -r` command.
- 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 via the `%macro` command. 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 via the `%autocall` command: 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.
- 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.
- Simple timing information. You can use the `%timeit` command to get the execution time of a Python statement or expression. This machinery is intelligent enough to do more repetitions for commands that finish very quickly in order to get a better estimate of their running time.

```
In [1]: %timeit 1+1
10000000 loops, best of 3: 25.5 ns per loop

In [2]: %timeit [math.sin(x) for x in range(5000)]
1000 loops, best of 3: 719 µs per loop
```

  To get the timing information for more than one expression, use the `%%timeit` cell magic command.

- Doctest support. The special `%doctest_mode` command toggles a mode to use doctest-compatible prompts, so you can use IPython sessions as doctest code. By default, IPython also allows you to paste existing doctests, and strips out the leading `>>>` and `...` prompts in them.

1.3 Decoupled two-process model

IPython has abstracted and extended the notion of a traditional *Read-Evaluate-Print Loop* (REPL) environment by decoupling the *evaluation* into its own process. We call this process a **kernel**: it receives execution instructions from clients and communicates the results back to them.

This decoupling allows us to have several clients connected to the same kernel, and even allows clients and kernels to live on different machines. With the exclusion of the traditional single process terminal-based IPython (what you start if you run `ipython` without any subcommands), all other IPython machinery uses this two-process model. This includes `ipython console`, `ipython qtconsole`, and `ipython notebook`.

As an example, this means that when you start `ipython qtconsole`, you're really starting two processes, a kernel and a Qt-based client can send commands to and receive results from that kernel. If there is already a kernel running that you want to connect to, you can pass the `--existing` flag which will skip initiating a new kernel and connect to the most recent kernel, instead. To connect to a specific kernel once you have several kernels running, use the `%connect_info` magic to get the unique connection file, which will be something like `--existing kernel-19732.json` but with different numbers which correspond to the Process ID of the kernel.

You can read more about using *ipython qtconsole*, and *ipython notebook*. There is also a *message spec* which documents the protocol for communication between kernels and clients.

See also:

Frontend/Kernel Model example notebook

1.4 Interactive parallel computing

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

The main features of this system are:

- Quickly parallelize Python code from an interactive Python/IPython session.
- A flexible and dynamic process model that be deployed on anything from multicore workstations to supercomputers.
- An architecture that supports many different styles of parallelism, from message passing to task farming. And all of these styles can be handled interactively.
- Both blocking and fully asynchronous interfaces.
- High level APIs that enable many things to be parallelized in a few lines of code.
- Write parallel code that will run unchanged on everything from multicore workstations to supercomputers.
- Full integration with Message Passing libraries (MPI).

- Capabilities based security model with full encryption of network connections.
- Share live parallel jobs with other users securely. We call this collaborative parallel computing.
- Dynamically load balanced task farming system.
- Robust error handling. Python exceptions raised in parallel execution are gathered and presented to the top-level code.

For more information, see our *overview* of using IPython for parallel computing.

1.4.1 Portability and Python requirements

As of the 1.0 release, IPython works with Python 2.6, 2.7, 3.2 and 3.3. Version 0.12 introduced full support for Python 3. Version 0.11 worked with Python 2.6 and 2.7 only. Versions 0.9 and 0.10 worked with Python 2.4 and above (not including Python 3).

IPython is known to work on the following operating systems:

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

See *here* for instructions on how to install IPython.

CHAPTER 2

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

This document describes in-flight development work.

2.1.1 Backwards incompatible changes

2.2 1.0 Series

2.2.1 Release 1.0.0: An Afternoon Hack

IPython 1.0 requires Python 2.6.5 or 3.2.1. It does not support Python 3.0, 3.1, or 2.5.

This is a big release. The principal milestone is the addition of `IPython.nbconvert`, but there has been a great deal of work improving all parts of IPython as well.

The previous version (0.13) was released on June 30, 2012, and in this development cycle we had:

- ~12 months of work.
- ~700 pull requests merged.
- ~600 issues closed (non-pull requests).
- contributions from ~150 authors.
- ~4000 commits.

The amount of work included in this release is so large that we can only cover here the main highlights; please see our *detailed release statistics* for links to every issue and pull request closed on GitHub as well as a full list of individual contributors. It includes

Reorganization

There have been two major reorganizations in IPython 1.0:

- Added `IPython.kernel` for all kernel-related code. This means that `IPython.zmq` has been removed, and much of it is now in `IPython.kernel.zmq`, some of it being in the top-level `IPython.kernel`.
- We have removed the *frontend* subpackage, as it caused unnecessary depth. So what was `IPython.frontend.qt` is now `IPython.qt`, and so on. The one difference is that the notebook has been further flattened, so that `IPython.frontend.html.notebook` is now just *IPython.html*. There is a shim module, so `IPython.frontend` is still importable in 1.0, but there will be a warning.
- The IPython sphinx directives are now installed in `IPython.sphinx`, so they can be imported by other projects.

Public APIs

For the first time since 0.10 (sorry, everyone), there is an official public API for starting IPython:

```
from IPython import start_ipython
start_ipython()
```

This is what packages should use that start their own IPython session, but don't actually want embedded IPython (most cases). `IPython.embed()` is used for embedding IPython into the calling namespace, similar to calling `Pdb.set_trace()`, whereas `start_ipython()` will start a plain IPython session, loading config and startup files as normal.

We also have added:

```
from IPython import get_ipython
```

Which is a *library* function for getting the current IPython instance, and will return `None` if no IPython instance is running. This is the official way to check whether your code is called from inside an IPython session. If you want to check for IPython without unnecessarily importing IPython, use this function:

```
def get_ipython():
    """return IPython instance if there is one, None otherwise"""
    import sys
    if "IPython" in sys.modules:
        import IPython
        return IPython.get_ipython()
```

Core

- The input transformation framework has been reworked. This fixes some corner cases, and adds more flexibility for projects which use IPython, like SymPy & SAGE. For more details, see Custom input transformation.

- Exception types can now be displayed with a custom traceback, by defining a `_render_traceback_()` method which returns a list of strings, each containing one line of the traceback.
- A new command, `ipython history trim` can be used to delete everything but the last 1000 entries in the history database.
- `__file__` is defined in both config files at load time, and `.ipy` files executed with `%run`.
- `%logstart` and `%logappend` are no longer broken.
- Add glob expansion for `%run`, e.g. `%run -g script.py *.txt`.
- Expand variables (`$foo`) in Cell Magic argument line.
- By default, **`iptest`** will exclude various slow tests. All tests can be run with **`iptest --all`**.
- SQLite history can be disabled in the various cases that it does not behave well.
- `%edit` works on interactively defined variables.
- editor hooks have been restored from quarantine, enabling TextMate as editor, etc.
- The env variable PYTHONSTARTUP is respected by IPython.
- The `%matplotlib` magic was added, which is like the old `%pylab` magic, but it does not import anything to the interactive namespace. It is recommended that users switch to `%matplotlib` and explicit imports.
- The `--matplotlib` command line flag was also added. It invokes the new `%matplotlib` magic and can be used in the same way as the old `--pylab` flag. You can either use it by itself as a flag (`--matplotlib`), or you can also pass a backend explicitly (`--matplotlib qt` or `--matplotlib=wx`, etc).

Backwards incompatible changes

- Calling `InteractiveShell.prefilter()` will no longer perform static transformations - the processing of escaped commands such as `%magic` and `!system`, and stripping input prompts from code blocks. This functionality was duplicated in `IPython.core.inputsplitter`, and the latter version was already what IPython relied on. A new API to transform input will be ready before release.
- Functions from `IPython.lib.inputhook` to control integration with GUI event loops are no longer exposed in the top level of `IPython.lib`. Code calling these should make sure to import them from `IPython.lib.inputhook`.
- For all kernel managers, the `sub_channel` attribute has been renamed to `iopub_channel`.
- Users on Python versions before 2.6.6, 2.7.1 or 3.2 will now need to call `IPython.utils.doctestreload.doctest_reload()` to make doctests run correctly inside IPython. Python releases since those versions are unaffected. For details, see PR #3068 and Python issue 8048.

- The `InteractiveShell.cache_main_mod()` method has been removed, and `new_main_mod()` has a different signature, expecting a filename where earlier versions expected a namespace. See PR #3555 for details.
- The short-lived plugin system has been removed. Extensions are the way to go.

NbConvert

The major milestone for IPython 1.0 is the addition of `IPython.nbconvert` - tools for converting IPython notebooks to various other formats.

Warning: nbconvert is α-level preview code in 1.0

To use nbconvert to convert various file formats:

```
ipython nbconvert --to html *.ipynb
```

See `ipython nbconvert --help` for more information. nbconvert depends on pandoc for many of the translations to and from various formats.

See also:

Converting notebooks to other formats

Notebook

Major changes to the IPython Notebook in 1.0:

- The notebook is now autosaved, by default at an interval of two minutes. When you press 'save' or Ctrl-S, a *checkpoint* is made, in a hidden folder. This checkpoint can be restored, so that the autosave model is strictly safer than traditional save. If you change nothing about your save habits, you will always have a checkpoint that you have written, and an autosaved file that is kept up to date.
- The notebook supports `raw_input()` / `input()`, and thus also `%debug`, and many other Python calls that expect user input.
- You can load custom javascript and CSS in the notebook by editing the files `$(ipython locate profile)/static/custom/custom.`*`js,css`*.
- Add `%%html`, `%%svg`, `%%javascript`, and `%%latex` cell magics for writing raw output in notebook cells.
- add a redirect handler and anchors on heading cells, so you can link across notebooks, directly to heading cells in other notebooks.
- Images support width and height metadata, and thereby 2x scaling (retina support).
- `_repr_foo_` methods can return a tuple of (data, metadata), where metadata is a dict containing metadata about the displayed object. This is used to set size, etc. for retina graphics. To enable retina matplotlib figures, simply set `InlineBackend.figure_format = 'retina'` for 2x PNG figures, in your *IPython config file* or via the `%config` magic.

- Add display.FileLink and FileLinks for quickly displaying HTML links to local files.
- Cells have metadata, which can be edited via cell toolbars. This metadata can be used by external code (e.g. reveal.js or exporters), when examining the notebook.
- Fix an issue parsing LaTeX in markdown cells, which required users to type `\\\`, instead of `\\`.
- Notebook templates are rendered with Jinja instead of Tornado.
- `%%file` has been renamed `%%writefile` (`%%file` is deprecated).
- ANSI (and VT100) color parsing has been improved in both performance and supported values.
- The static files path can be found as `IPython.html.DEFAULT_STATIC_FILES_PATH`, which may be changed by package managers.
- IPython's CSS is installed in `static/css/style.min.css` (all style, including bootstrap), and `static/css/ipython.min.css`, which only has IPython's own CSS. The latter file should be useful for embedding IPython notebooks in other pages, blogs, etc.
- The Print View has been removed. Users are encouraged to test *ipython nbconvert* to generate a static view.

Javascript Components

The javascript components used in the notebook have been updated significantly.

- updates to jQuery (2.0) and jQueryUI (1.10)
- Update CodeMirror to 3.14
- Twitter Bootstrap (2.3) for layout
- Font-Awesome (3.1) for icons
- highlight.js (7.3) for syntax highlighting
- marked (0.2.8) for markdown rendering
- require.js (2.1) for loading javascript

Some relevant changes that are results of this:

- Markdown cells now support GitHub-flavored Markdown (GFM), which includes ` ```python ` code blocks and tables.
- Notebook UI behaves better on more screen sizes.
- Various code cell input issues have been fixed.

Kernel

The kernel code has been substantially reorganized.

New features in the kernel:

- Kernels support ZeroMQ IPC transport, not just TCP

- The message protocol has added a top-level metadata field, used for information about messages.
- Add a *data_pub* message that functions much like *display_pub*, but publishes raw (usually pickled) data, rather than representations.
- Ensure that `sys.stdout.encoding` is defined in Kernels.
- Stdout from forked subprocesses should be forwarded to frontends (instead of crashing).

IPEP 13

The KernelManager has been split into a `KernelManager` and a `KernelClient`. The Manager owns a kernel and starts / signals / restarts it. There is always zero or one KernelManager per Kernel. Clients communicate with Kernels via zmq channels, and there can be zero-to-many Clients connected to a Kernel at any given time.

The KernelManager now automatically restarts the kernel when it dies, rather than requiring user input at the notebook or QtConsole UI (which may or may not exist at restart time).

In-process kernels

The Python-language frontends, particularly the Qt console, may now communicate with in-process kernels, in addition to the traditional out-of-process kernels. An in-process kernel permits direct access to the kernel namespace, which is necessary in some applications. It should be understood, however, that the in-process kernel is not robust to bad user input and will block the main (GUI) thread while executing. Developers must decide on a case-by-case basis whether this tradeoff is appropriate for their application.

Parallel

IPython.parallel has had some refactoring as well. There are many improvements and fixes, but these are the major changes:

- Connections have been simplified. All ports and the serialization in use are written to the connection file, rather than the initial two-stage system.
- Serialization has been rewritten, fixing many bugs and dramatically improving performance serializing large containers.
- Load-balancing scheduler performance with large numbers of tasks has been dramatically improved.
- There should be fewer (hopefully zero) false-positives for engine failures.
- Increased compatibility with various use cases that produced serialization / argument errors with map, etc.
- The controller can attempt to resume operation if it has crashed, by passing `ipcontroller --restore`.
- Engines can monitor the Hub heartbeat, and shutdown if the Hub disappears for too long.
- add HTCondor support in launchers

QtConsole

Various fixes, including improved performance with lots of text output, and better drag and drop support. The initial window size of the qtconsole is now configurable via `IPythonWidget.width` and `IPythonWidget.height`.

2.3 Issues closed in the 1.0 development cycle

2.3.1 Issues closed in 1.2

GitHub stats for 2013/09/09 - 2014/02/21

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 13 authors contributed 84 commits.

- Benjamin Ragan-Kelley
- Daryl Herzmann
- Doug Blank
- Fernando Perez
- James Porter
- Juergen Hasch
- Julian Taylor
- Kyle Kelley
- Lawrence Fu
- Matthias Bussonnier
- Paul Ivanov
- Pascal Schetelat
- Puneeth Chaganti
- Takeshi Kanmae
- Thomas Kluyver

We closed a total of 55 issues, 38 pull requests and 17 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (38):

1.2.1:

- PR #4372: Don't assume that SyntaxTB is always called with a SyntaxError
- PR #5166: remove mktemp usage

- PR #5163: Simplify implementation of TemporaryWorkingDirectory.
- PR #5105: add index to format to support py2.6

1.2.0:

- PR #4972: Work around problem in doctest discovery in Python 3.4 with PyQt
- PR #4934: *ipython profile create* respects *–ipython-dir*
- PR #4845: Add Origin Checking.
- PR #4928: use importlib.machinery when available
- PR #4849: Various unicode fixes (mostly on Windows)
- PR #4880: set profile name from profile_dir
- PR #4908: detect builtin docstrings in oinspect
- PR #4909: sort dictionary keys before comparison, ordering is not guaranteed
- PR #4903: use https for all embeds
- PR #4868: Static path fixes
- PR #4820: fix regex for cleaning old logs with ipcluster
- PR #4840: Error in Session.send_raw()
- PR #4762: whitelist alphanumeric characters for cookie_name
- PR #4748: fix race condition in profiledir creation.
- PR #4720: never use ssh multiplexer in tunnels
- PR #4738: don’t inject help into user_ns
- PR #4722: allow purging local results as long as they are not outstanding
- PR #4668: Make non-ASCII docstring unicode
- PR #4639: Minor import fix to get qtconsole with –pylab=qt working
- PR #4453: Play nice with App Nap
- PR #4609: Fix bytes regex for Python 3.
- PR #4488: fix typo in message spec doc
- PR #4346: getpass() on Windows & Python 2 needs bytes prompt
- PR #4230: Switch correctly to the user’s default matplotlib backend after inline.
- PR #4214: engine ID metadata should be unicode, not bytes
- PR #4232: no highlight if no language specified
- PR #4218: Fix display of SyntaxError when .py file is modified
- PR #4217: avoid importing numpy at the module level
- PR #4213: fixed dead link in examples/notebooks readme to Part 3

- PR #4183: ESC should be handled by CM if tooltip is not on
- PR #4193: Update for #3549: Append Firefox overflow-x fix
- PR #4205: use TextIOWrapper when communicating with pandoc subprocess
- PR #4204: remove some extraneous print statements from IPython.parallel
- PR #4201: HeadingCells cannot be split or merged

Issues (17):

1.2.1:

- #5101: IPython 1.2.0: notebook fail with "500 Internal Server Error"

1.2.0:

- #4892: IPython.qt test failure with python3.4
- #4810: ipcluster bug in clean_logs flag
- #4765: missing build script for highlight.js
- #4761: ipv6 address triggers cookie exception
- #4721: purge_results with jobid crashing - looking for insight
- #4602: "ipcluster stop" fails after "ipcluster start –daemonize" using python3.3
- #3386: Magic %paste not working in Python 3.3.2. TypeError: Type str doesn't support the buffer API
- #4485: Incorrect info in "Messaging in IPython" documentation.
- #4351: /parallel/apps/launcher.py error
- #4334: NotebookApp.webapp_settings static_url_prefix causes crash
- #4039: Celltoolbar example issue
- #4256: IPython no longer handles unicode file names
- #4122: Nbconvert [windows]: Inconsistent line endings in markdown cells exported to latex
- #3819: nbconvert add extra blank line to code block on Windows.
- #4203: remove spurious print statement from parallel annoted functions
- #4200: Notebook: merging a heading cell and markdown cell cannot be undone

2.3.2 Issues closed in 1.1

GitHub stats for 2013/08/08 - 2013/09/09 (since 1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 25 authors contributed 337 commits.

- Benjamin Ragan-Kelley

- Bing Xia
- Bradley M. Froehle
- Brian E. Granger
- Damián Avila
- dhirschfeld
- Dražen Lučanin
- gmbecker
- Jake Vanderplas
- Jason Grout
- Jonathan Frederic
- Kevin Burke
- Kyle Kelley
- Matt Henderson
- Matthew Brett
- Matthias Bussonnier
- Pankaj Pandey
- Paul Ivanov
- rossant
- Samuel Ainsworth
- Stephan Rave
- stonebig
- Thomas Kluyver
- Yaroslav Halchenko
- Zachary Sailer

We closed a total of 76 issues, 58 pull requests and 18 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (58):

- PR #4188: Allow user_ns trait to be None
- PR #4189: always fire LOCAL_IPS.extend(PUBLIC_IPS)
- PR #4174: various issues in markdown and rst templates
- PR #4178: add missing data_javascript
- PR #4181: nbconvert: Fix, sphinx template not removing new lines from headers

- PR #4043: don't 'restore_bytes' in from_JSON
- PR #4163: Fix for incorrect default encoding on Windows.
- PR #4136: catch javascript errors in any output
- PR #4171: add nbconvert config file when creating profiles
- PR #4125: Basic exercise of *ipython [subcommand] -h* and help-all
- PR #4085: nbconvert: Fix sphinx preprocessor date format string for Windows
- PR #4159: don't split *.cell* and *div.cell* CSS
- PR #4158: generate choices for *–gui* configurable from real mapping
- PR #4065: do not include specific css in embedable one
- PR #4092: nbconvert: Fix for unicode html headers, Windows + Python 2.x
- PR #4074: close Client sockets if connection fails
- PR #4064: Store default codemirror mode in only 1 place
- PR #4104: Add way to install MathJax to a particular profile
- PR #4144: help_end transformer shouldn't pick up ? in multiline string
- PR #4143: update example custom.js
- PR #4142: DOC: unwrap openssl line in public_server doc
- PR #4141: add files with a separate *add* call in backport_pr
- PR #4137: Restore autorestore option for storemagic
- PR #4098: pass profile-dir instead of profile name to Kernel
- PR #4120: support *input* in Python 2 kernels
- PR #4088: nbconvert: Fix coalescestreams line with incorrect nesting causing strange behavior
- PR #4060: only strip continuation prompts if regular prompts seen first
- PR #4132: Fixed name error bug in function safe_unicode in module py3compat.
- PR #4121: move test_kernel from IPython.zmq to IPython.kernel
- PR #4118: ZMQ heartbeat channel: catch EINTR exceptions and continue.
- PR #4054: use unicode for HTML export
- PR #4106: fix a couple of default block values
- PR #4115: Update docs on declaring a magic function
- PR #4101: restore accidentally removed EngineError
- PR #4096: minor docs changes
- PR #4056: respect *pylab_import_all* when *–pylab* specified at the command-line
- PR #4091: Make Qt console banner configurable

- PR #4086: fix missing errno import
- PR #4030: exclude *.git* in MANIFEST.in
- PR #4047: Use istype() when checking if canned object is a dict
- PR #4031: don't close_fds on Windows
- PR #4029: bson.Binary moved
- PR #4035: Fixed custom jinja2 templates being ignored when setting template_path
- PR #4026: small doc fix in nbconvert
- PR #4016: Fix IPython.start_* functions
- PR #4021: Fix parallel.client.View map() on numpy arrays
- PR #4022: DOC: fix links to matplotlib, notebook docs
- PR #4018: Fix warning when running IPython.kernel tests
- PR #4019: Test skipping without unicode paths
- PR #4008: Transform code before %prun/%%prun runs
- PR #4014: Fix typo in ipapp
- PR #3987: get files list in backport_pr
- PR #3974: nbconvert: Fix app tests on Window7 w/ Python 3.3
- PR #3978: fix *–existing* with non-localhost IP
- PR #3939: minor checkpoint cleanup
- PR #3981: BF: fix nbconvert rst input prompt spacing
- PR #3960: Don't make sphinx a dependency for importing nbconvert
- PR #3973: logging.Formatter is not new-style in 2.6

Issues (18):

- #4024: nbconvert markdown issues
- #4095: Catch js error in append html in stream/pyerr
- #4156: Specifying –gui=tk at the command line
- #3818: nbconvert can't handle Heading with Chinese characters on Japanese Windows OS.
- #4134: multi-line parser fails on '" in comment, qtconsole and notebook.
- #3998: sample custom.js needs to be updated
- #4078: StoreMagic.autorestore not working in 1.0.0
- #3990: Buitlin *input* doesn't work over zmq
- #4015: nbconvert fails to convert all the content of a notebook
- #4059: Issues with Ellipsis literal in Python 3

- #4103: Wrong default argument of DirectView.clear
- #4100: parallel.client.client references undefined error.EngineError
- #4005: IPython.start_kernel doesn't work.
- #4020: IPython parallel map fails on numpy arrays
- #3945: nbconvert: commandline tests fail Win7x64 Py3.3
- #3977: unable to complete remote connections for two-process
- #3980: nbconvert rst output lacks needed blank lines
- #3968: TypeError: super() argument 1 must be type, not classobj (Python 2.6.6)

2.3.3 Issues closed in 1.0

GitHub stats for 2012/06/30 - 2013/08/08 (since 0.13)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 155 authors contributed 4258 commits.

- Aaron Meurer
- Adam Davis
- Ahmet Bakan
- Alberto Valverde
- Allen Riddell
- Anders Hovmöller
- Andrea Bedini
- Andrew Spiers
- Andrew Vandever
- Anthony Scopatz
- Anton Akhmerov
- Anton I. Sipos
- Antony Lee
- Aron Ahmadia
- Benedikt Sauer
- Benjamin Jones
- Benjamin Ragan-Kelley
- Benjie Chen
- Boris de Laage

- Brad Reisfeld
- Bradley M. Froehle
- Brian E. Granger
- Cameron Bates
- Cavendish McKay
- chapmanb
- Chris Beaumont
- Chris Laumann
- Christoph Gohlke
- codebraker
- codespaced
- Corran Webster
- DamianHeard
- Damián Avila
- Dan Kilman
- Dan McDougall
- Danny Staple
- David Hirschfeld
- David P. Sanders
- David Warde-Farley
- David Wolever
- David Wyde
- debjan
- Diane Trout
- dkua
- Dominik Dabrowski
- Donald Curtis
- Dražen Lučanin
- drevicko
- Eric O. LEBIGOT
- Erik M. Bray
- Erik Tollerud

- Eugene Van den Bulke
- Evan Patterson
- Fernando Perez
- Francesco Montesano
- Frank Murphy
- Greg Caporaso
- Guy Haskin Fernald
- guziy
- Hans Meine
- Harry Moreno
- henryiii
- Ivan Djokic
- Jack Feser
- Jake Vanderplas
- jakobgager
- James Booth
- Jan Schulz
- Jason Grout
- Jeff Knisley
- Jens Hedegaard Nielsen
- jeremiahbuddha
- Jerry Fowler
- Jessica B. Hamrick
- Jez Ng
- John Zwinck
- Jonathan Frederic
- Jonathan Taylor
- Joon Ro
- Joseph Lansdowne
- Juergen Hasch
- Julian Taylor
- Jussi Sainio

- Jörgen Stenarson
- kevin
- klonuo
- Konrad Hinsen
- Kyle Kelley
- Lars Solberg
- Lessandro Mariano
- Mark Sienkiewicz at STScI
- Martijn Vermaat
- Martin Spacek
- Matthias Bussonnier
- Maxim Grechkin
- Maximilian Albert
- MercuryRising
- Michael Droettboom
- Michael Shuffett
- Michał Górny
- Mikhail Korobov
- mr.Shu
- Nathan Goldbaum
- ocefpaf
- Ohad Ravid
- Olivier Grisel
- Olivier Verdier
- Owen Healy
- Pankaj Pandey
- Paul Ivanov
- Pawel Jasinski
- Pietro Berkes
- Piti Ongmongkolkul
- Puneeth Chaganti
- Rich Wareham

- Richard Everson
- Rick Lupton
- Rob Young
- Robert Kern
- Robert Marchman
- Robert McGibbon
- Rui Pereira
- Rustam Safin
- Ryan May
- s8weber
- Samuel Ainsworth
- Sean Vig
- Siyu Zhang
- Skylar Saveland
- slojo404
- smithj1
- Stefan Karpinski
- Stefan van der Walt
- Steven Silvester
- Takafumi Arakaki
- Takeshi Kanmae
- tcmulcahy
- teegaar
- Thomas Kluyver
- Thomas Robitaille
- Thomas Spura
- Thomas Weißschuh
- Timothy O'Donnell
- Tom Dimiduk
- ugurthemaster
- urielshaolin
- v923z

- Valentin Haenel
- Victor Zverovich
- 23. Trevor King
- y-p
- Yoav Ram
- Zbigniew Jędrzejewski-Szmek
- Zoltán Vörös

We closed a total of 1484 issues, 793 pull requests and 691 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (793):

- PR #3958: doc update
- PR #3965: Fix ansi color code for background yellow
- PR #3964: Fix casing of message.
- PR #3942: Pass on install docs
- PR #3962: exclude IPython.lib.kernel in iptest
- PR #3961: Longpath test fix
- PR #3905: Remove references to 0.11 and 0.12 from config/overview.rst
- PR #3951: nbconvert: fixed latex characters not escaped properly in nbconvert
- PR #3949: log fatal error when PDF conversion fails
- PR #3947: nbconvert: Make writer & post-processor aliases case insensitive.
- PR #3938: Recompile css.
- PR #3948: sphinx and PDF tweaks
- PR #3943: nbconvert: Serve post-processor Windows fix
- PR #3934: nbconvert: fix logic of verbose flag in PDF post processor
- PR #3929: swallow enter event in rename dialog
- PR #3924: nbconvert: Backport fixes
- PR #3925: Replace –pylab flag with –matplotlib in usage
- PR #3910: Added explicit error message for missing configuration arguments.
- PR #3913: grffile to support spaces in notebook names
- PR #3918: added check_for_tornado, closes #3916
- PR #3917: change docs/examples refs to be just examples
- PR #3908: what's new tweaks

- PR #3896: two column quickhelp dialog, closes #3895
- PR #3911: explicitly load python mode before IPython mode
- PR #3901: don't force . relative path, fix #3897
- PR #3891: fix #3889
- PR #3892: Fix documentation of Kernel.stop_channels
- PR #3888: posixify paths for Windows latex
- PR #3882: quick fix for #3881
- PR #3877: don't use *shell=True* in PDF export
- PR #3878: minor template loading cleanup
- PR #3855: nbconvert: Filter tests
- PR #3879: finish 3870
- PR #3870: Fix for converting notebooks that contain unicode characters.
- PR #3876: Update parallel_winhpc.rst
- PR #3872: removing vim-ipython, since it has it's own repo
- PR #3871: updating docs
- PR #3873: remove old examples
- PR #3868: update CodeMirror component to 3.15
- PR #3865: Escape filename for pdflatex in nbconvert
- PR #3861: remove old external.js
- PR #3864: add keyboard shortcut to docs
- PR #3834: This PR fixes a few issues with nbconvert tests
- PR #3840: prevent profile_dir from being undefined
- PR #3859: Add "An Afternoon Hack" to docs
- PR #3854: Catch errors filling readline history on startup
- PR #3857: Delete extra auto
- PR #3845: nbconvert: Serve from original build directory
- PR #3846: Add basic logging to nbconvert
- PR #3850: add missing store_history key to Notebook execute_requests
- PR #3844: update payload source
- PR #3830: mention metadata / display_data similarity in pyout spec
- PR #3848: fix incorrect *empty-docstring*
- PR #3836: Parse markdown correctly when mathjax is disabled

- PR #3849: skip a failing test on windows
- PR #3828: signature_scheme lives in Session
- PR #3831: update nbconvert doc with new CLI
- PR #3822: add output flag to nbconvert
- PR #3780: Added serving the output directory if html-based format are selected.
- PR #3764: Cleanup nbconvert templates
- PR #3829: remove now-duplicate 'this is dev' note
- PR #3814: add *ConsoleWidget.execute_on_complete_input* flag
- PR #3826: try rtfd
- PR #3821: add sphinx prolog
- PR #3817: relax timeouts in terminal console and tests
- PR #3825: fix more tests that fail when pandoc is missing
- PR #3824: don't set target on internal markdown links
- PR #3816: s/pylab/matplotlib in docs
- PR #3812: Describe differences between start_ipython and embed
- PR #3805: Print View has been removed
- PR #3820: Make it clear that 1.0 is not released yet
- PR #3784: nbconvert: Export flavors & PDF writer (ipy dev meeting)
- PR #3800: semantic-versionify version number for non-releases
- PR #3802: Documentation .txt to .rst
- PR #3765: cleanup terminal console iopub handling
- PR #3720: Fix for #3719
- PR #3787: re-raise KeyboardInterrupt in raw_input
- PR #3770: Organizing reveal's templates.
- PR #3751: Use link(2) when possible in nbconvert
- PR #3792: skip tests that require pandoc
- PR #3782: add Importing Notebooks example
- PR #3752: nbconvert: Add cwd to sys.path
- PR #3789: fix raw_input in qtconsole
- PR #3756: document the wire protocol
- PR #3749: convert IPython syntax to Python syntax in nbconvert python template
- PR #3793: Closes #3788

- PR #3794: Change logo link to ipython.org
- PR #3746: Raise a named exception when pandoc is missing
- PR #3781: comply with the message spec in the notebook
- PR #3779: remove bad *if logged_in* preventing new-notebook without login
- PR #3743: remove notebook read-only view
- PR #3732: add delay to autosave in beforeunload
- PR #3761: Added rm_math_space to markdown cells in the basichtml.tpl to be rendered ok by mathjax after the nbconvertion.
- PR #3758: nbconvert: Filter names cleanup
- PR #3769: Add configurability to tabcompletion timeout
- PR #3771: Update px pylab test to match new output of pylab
- PR #3741: better message when notebook format is not supported
- PR #3753: document Ctrl-C not working in ipython kernel
- PR #3766: handle empty metadata in pyout messages more gracefully.
- PR #3736: my attempt to fix #3735
- PR #3759: nbconvert: Provide a more useful error for invalid use case.
- PR #3760: nbconvert: Allow notebook filenames without their extensions
- PR #3750: nbconvert: Add cwd to default templates search path.
- PR #3748: Update nbconvert docs
- PR #3734: Nbconvert: Export extracted files into *nbname_files* subdirectory
- PR #3733: Nicer message when pandoc is missing, closes #3730
- PR #3722: fix two failing test in IPython.lib
- PR #3704: Start what's new for 1.0
- PR #3705: Complete rewrite of IPython Notebook documentation: docs/source/interactive/htmlnotebook.txt
- PR #3709: Docs cleanup
- PR #3716: raw_input fixes for kernel restarts
- PR #3683: use *%matplotlib* in example notebooks
- PR #3686: remove quarantine
- PR #3699: svg2pdf unicode fix
- PR #3695: fix SVG2PDF
- PR #3685: fix Pager.detach
- PR #3675: document new dependencies

- PR #3690: Fixing some css minors in full_html and reveal.
- PR #3671: nbconvert tests
- PR #3692: Fix rename notebook - show error with invalid name
- PR #3409: Prevent qtconsole frontend freeze on lots of output.
- PR #3660: refocus active cell on dialog close
- PR #3598: Statelessify mathjaxutils
- PR #3673: enable comment/uncomment selection
- PR #3677: remove special-case in get_home_dir for frozen dists
- PR #3674: add CONTRIBUTING.md
- PR #3670: use Popen command list for ipexec
- PR #3568: pylab import adjustments
- PR #3559: add create.Cell and delete.Cell js events
- PR #3606: push cell magic to the head of the transformer line
- PR #3607: NbConvert: Writers, No YAML, and stuff...
- PR #3665: Pywin32 skips
- PR #3669: set default client_class for QtKernelManager
- PR #3662: add strip_encoding_cookie transformer
- PR #3641: increase patience for slow kernel startup in tests
- PR #3651: remove a bunch of unused *default_config_file* assignments
- PR #3630: CSS adjustments
- PR #3645: Don't require HistoryManager to have a shell
- PR #3643: don't assume tested ipython is on the PATH
- PR #3654: fix single-result AsyncResults
- PR #3601: Markdown in heading cells (take 2)
- PR #3652: Remove old *docs/examples*
- PR #3621: catch any exception appending output
- PR #3585: don't blacklist builtin names
- PR #3647: Fix *frontend* deprecation warnings in several examples
- PR #3649: fix AsyncResult.get_dict for single result
- PR #3648: Fix store magic test
- PR #3650: Fix, config_file_name was ignored
- PR #3640: Gcf.get_active() can return None

- PR #3571: Added shorcuts to split cell, merge cell above and merge cell below.
- PR #3635: Added missing slash to print-pdf call.
- PR #3487: Drop patch for compatibility with pyreadline 1.5
- PR #3338: Allow filename with extension in find_cmd in Windows.
- PR #3628: Fix test for Python 3 on Windows.
- PR #3642: Fix typo in docs
- PR #3627: use DEFAULT_STATIC_FILES_PATH in a test instead of package dir
- PR #3624: fix some unicode in zmqhandlers
- PR #3460: Set calling program to UNKNOWN, when argv not in sys
- PR #3632: Set calling program to UNKNOWN, when argv not in sys (take #2)
- PR #3629: Use new entry point for python -m IPython
- PR #3626: passing cell to showInPager, closes #3625
- PR #3618: expand terminal color support
- PR #3623: raise UsageError for unsupported GUI backends
- PR #3071: Add magic function %drun to run code in debugger
- PR #3608: a nicer error message when using %pylab magic
- PR #3592: add extra_config_file
- PR #3612: updated .mailmap
- PR #3616: Add examples for interactive use of MPI.
- PR #3615: fix regular expression for ANSI escapes
- PR #3586: Corrected a typo in the format string for strftime the sphinx.py transformer of nbconvert
- PR #3611: check for markdown no longer needed, closes #3610
- PR #3555: Simplify caching of modules with %run
- PR #3583: notebook small things
- PR #3594: Fix duplicate completion in notebook
- PR #3600: parallel: Improved logging for errors during BatchSystemLauncher.stop
- PR #3595: Revert "allow markdown in heading cells"
- PR #3538: add IPython.start_ipython
- PR #3562: Allow custom nbconvert template loaders
- PR #3582: pandoc adjustments
- PR #3560: Remove max_msg_size
- PR #3591: Refer to Setuptools instead of Distribute

- PR #3590: IPython.sphinxext needs an __init__.py
- PR #3581: Added the possibility to read a custom.css file for tweaking the final html in full_html and reveal templates.
- PR #3576: Added support for markdown in heading cells when they are nbconverted.
- PR #3575: tweak *run -d* message to 'continue execution'
- PR #3569: add PYTHONSTARTUP to startup files
- PR #3567: Trigger a single event on js app initilized
- PR #3565: style.min.css shoudl always exist...
- PR #3531: allow markdown in heading cells
- PR #3577: Simplify codemirror ipython-mode
- PR #3495: Simplified regexp, and suggestions for clearer regexps.
- PR #3578: Use adjustbox to specify figure size in nbconvert -> latex
- PR #3572: Skip import irunner test on Windows.
- PR #3574: correct static path for CM modes autoload
- PR #3558: Add IPython.sphinxext
- PR #3561: mention double-control-C to stop notebook server
- PR #3566: fix event names
- PR #3564: Remove trivial nbconvert example
- PR #3540: allow cython cache dir to be deleted
- PR #3527: cleanup stale, unused exceptions in parallel.error
- PR #3529: ensure raw_input returns str in zmq shell
- PR #3541: respect image size metadata in qtconsole
- PR #3550: Fixing issue preventing the correct read of images by full_html and reveal exporters.
- PR #3557: open markdown links in new tabs
- PR #3556: remove mention of nonexistent *_margv* in macro
- PR #3552: set overflow-x: hidden on Firefox only
- PR #3554: Fix missing import os in latex exporter.
- PR #3546: Don't hardcode **latex** posix paths in nbconvert
- PR #3551: fix path prefix in nbconvert
- PR #3533: Use a CDN to get reveal.js library.
- PR #3498: When a notebook is written to file, name the metadata name u''.
- PR #3548: Change to standard save icon in Notebook toolbar

- PR #3539: Don't hardcode posix paths in nbconvert
- PR #3508: notebook supports raw_input and %debug now
- PR #3526: ensure 'default' is first in cluster profile list
- PR #3525: basic timezone info
- PR #3532: include nbconvert templates in installation
- PR #3515: update CodeMirror component to 3.14
- PR #3513: add 'No Checkpoints' to Revert menu
- PR #3536: format positions are required in Python 2.6.x
- PR #3521: Nbconvert fix, silent fail if template doesn't exist
- PR #3530: update %store magic docstring
- PR #3528: fix local mathjax with custom base_project_url
- PR #3518: Clear up unused imports
- PR #3506: %store -r restores saved aliases and directory history, as well as variables
- PR #3516: make css highlight style configurable
- PR #3523: Exclude frontend shim from docs build
- PR #3514: use bootstrap *disabled* instead of *ui-state-disabled*
- PR #3520: Added relative import of RevealExporter to __init__.py inside exporters module
- PR #3507: fix HTML capitalization in nbconvert exporter classes
- PR #3512: fix nbconvert filter validation
- PR #3511: Get Tracer working after ipapi.get replaced with get_ipython
- PR #3510: use *window.onbeforeunload=* for nav-away warning
- PR #3504: don't use parent=self in handlers
- PR #3500: Merge nbconvert into IPython
- PR #3478: restore "unsaved changes" warning on unload
- PR #3493: add a dialog when the kernel is auto-restarted
- PR #3488: Add test suite for autoreload extension
- PR #3484: Catch some pathological cases inside oinspect
- PR #3481: Display R errors without Python traceback
- PR #3468: fix *%magic* output
- PR #3430: add parent to Configurable
- PR #3491: Remove unexpected keyword parameter to remove_kernel
- PR #3485: SymPy has changed its recommended way to initialize printing

- PR #3486: Add test for non-ascii characters in docstrings
- PR #3483: Inputtransformer: Allow classic prompts without space
- PR #3482: Use an absolute path to iptest, because the tests are not always run from $IPYTHONDIR.
- PR #3381: enable 2x (retina) display
- PR #3450: Flatten IPython.frontend
- PR #3477: pass config to subapps
- PR #3466: Kernel fails to start when username has non-ascii characters
- PR #3465: Add HTCondor bindings to IPython.parallel
- PR #3463: fix typo, closes #3462
- PR #3456: Notice for users who disable javascript
- PR #3453: fix cell execution in firefox, closes #3447
- PR #3393: [WIP] bootstrapify
- PR #3440: Fix installing mathjax from downloaded file via command line
- PR #3431: Provide means for starting the Qt console maximized and with the menu bar hidden
- PR #3425: base IPClusterApp inherits from BaseIPythonApp
- PR #3433: Update IPythonexternalpath__init__.py
- PR #3298: Some fixes in IPython Sphinx directive
- PR #3428: process escapes in mathjax
- PR #3420: thansk -> thanks
- PR #3416: Fix doc: "principle" not "principal"
- PR #3413: more unique filename for test
- PR #3364: Inject requirejs in notebook and start using it.
- PR #3390: Fix %paste with blank lines
- PR #3403: fix creating config objects from dicts
- PR #3401: rollback #3358
- PR #3373: make cookie_secret configurable
- PR #3307: switch default ws_url logic to js side
- PR #3392: Restore anchor link on h2-h6
- PR #3369: Use different treshold for (auto)scroll in output
- PR #3370: normalize unicode notebook filenames
- PR #3372: base default cookie name on request host+port
- PR #3378: disable CodeMirror drag/drop on Safari

- PR #3358: workaround spurious CodeMirror scrollbars
- PR #3371: make setting the notebook dirty flag an event
- PR #3366: remove long-dead zmq frontend.py and completer.py
- PR #3382: cull Session digest history
- PR #3330: Fix get_ipython_dir when $HOME is /
- PR #3319: IPEP 13: user-expressions and user-variables
- PR #3384: comments in tools/gitwash_dumper.py changed (‘” to “””)
- PR #3387: Make submodule checks work under Python 3.
- PR #3357: move anchor-link off of heading text
- PR #3351: start basic tests of ipcluster Launchers
- PR #3377: allow class.__module__ to be None
- PR #3340: skip submodule check in package managers
- PR #3328: decode subprocess output in launchers
- PR #3368: Reenable bracket matching
- PR #3356: Mpr fixes
- PR #3336: Use new input transformation API in %time magic
- PR #3325: Organize the JS and less files by component.
- PR #3342: fix test_find_cmd_python
- PR #3354: catch socket.error in utils.localinterfaces
- PR #3341: fix default cluster count
- PR #3286: don’t use *get_ipython* from builtins in library code
- PR #3333: notebookapp: add missing whitespace to warnings
- PR #3323: Strip prompts even if the prompt isn’t present on the first line.
- PR #3321: Reorganize the python/server side of the notebook
- PR #3320: define __*file*__ in config files
- PR #3317: rename *%%file* to *%%writefile*
- PR #3304: set unlimited HWM for all relay devices
- PR #3315: Update Sympy_printing extension load
- PR #3310: further clarify Image docstring
- PR #3285: load extensions in builtin trap
- PR #3308: Speed up AsyncResult._wait_for_outputs(0)
- PR #3294: fix callbacks as optional in js kernel.execute

- PR #3276: Fix: "python ABS/PATH/TO/ipython.py" fails
- PR #3301: allow python3 tests without python installed
- PR #3282: allow view.map to work with a few more things
- PR #3284: remove *ipython.py* entry point
- PR #3281: fix ignored IOPub messages with no parent
- PR #3275: improve submodule messages / git hooks
- PR #3239: Allow "x" icon and esc key to close pager in notebook
- PR #3290: Improved heartbeat controller to engine monitoring for long running tasks
- PR #3142: Better error message when CWD doesn't exist on startup
- PR #3066: Add support for relative import to %run -m (fixes #2727)
- PR #3269: protect highlight.js against unknown languages
- PR #3267: add missing return
- PR #3101: use marked / highlight.js instead of pagedown and prettify
- PR #3264: use https url for submodule
- PR #3263: fix set_last_checkpoint when no checkpoint
- PR #3258: Fix submodule location in setup.py
- PR #3254: fix a few URLs from previous PR
- PR #3240: remove js components from the repo
- PR #3158: IPEP 15: autosave the notebook
- PR #3252: move images out of _static folder into _images
- PR #3251: Fix for cell magics in Qt console
- PR #3250: Added a simple __html__() method to the HTML class
- PR #3249: remove copy of sphinx inheritance_diagram.py
- PR #3235: Remove the unused print notebook view
- PR #3238: Improve the design of the tab completion UI
- PR #3242: Make changes of Application.log_format effective
- PR #3219: Workaround so only one CTRL-C is required for a new prompt in –gui=qt
- PR #3190: allow formatters to specify metadata
- PR #3231: improve discovery of public IPs
- PR #3233: check prefixes for swallowing kernel args
- PR #3234: Removing old autogrow JS code.
- PR #3232: Update to CodeMirror 3 and start to ship our components

- PR #3229: The HTML output type accidentally got removed from the OutputArea.
- PR #3228: Typo in IPython.Parallel documentation
- PR #3226: Text in rename dialog was way too big - making it <p>.
- PR #3225: Removing old restuctured text handler and web service.
- PR #3222: make BlockingKernelClient the default Client
- PR #3223: add missing mathjax_url to new settings dict
- PR #3089: add stdin to the notebook
- PR #3221: Remove references to HTMLCell (dead code)
- PR #3205: add ignored *args to HasTraits constructor
- PR #3088: cleanup IPython handler settings
- PR #3201: use much faster regexp for ansi coloring
- PR #3220: avoid race condition in profile creation
- PR #3011: IPEP 12: add KernelClient
- PR #3217: informative error when trying to load directories
- PR #3174: Simple class
- PR #2979: CM configurable Take 2
- PR #3215: Updates storemagic extension to allow for specifying variable name to load
- PR #3181: backport If-Modified-Since fix from tornado
- PR #3200: IFrame (VimeoVideo, ScribdDocument, ...)
- PR #3186: Fix small inconsistency in nbconvert: etype -> ename
- PR #3212: Fix issue #2563, "core.profiledir.check_startup_dir() doesn't work inside py2exe'd installation"
- PR #3211: Fix inheritance_diagram Sphinx extension for Sphinx 1.2
- PR #3208: Update link to extensions index
- PR #3203: Separate InputSplitter for transforming whole cells
- PR #3189: Improve completer
- PR #3194: finish up PR #3116
- PR #3188: Add new keycodes
- PR #2695: Key the root modules cache by sys.path entries.
- PR #3182: clarify %%file docstring
- PR #3163: BUG: Fix the set and frozenset pretty printer to handle the empty case correctly
- PR #3180: better UsageError for cell magic with no body

- PR #3184: Cython cache
- PR #3175: Added missing s
- PR #3173: Little bits of documentation cleanup
- PR #2635: Improve Windows start menu shortcuts (#2)
- PR #3172: Add missing import in IPython parallel magics example
- PR #3170: default application logger shouldn't propagate
- PR #3159: Autocompletion for zsh
- PR #3105: move DEFAULT_STATIC_FILES_PATH to IPython.html
- PR #3144: minor bower tweaks
- PR #3141: Default color output for ls on OSX
- PR #3137: fix dot syntax error in inheritance diagram
- PR #3072: raise UnsupportedOperation on iostream.fileno()
- PR #3147: Notebook support for a reverse proxy which handles SSL
- PR #3152: make qtconsole size at startup configurable
- PR #3162: adding stream kwarg to current.new_output
- PR #2981: IPEP 10: kernel side filtering of display formats
- PR #3058: add redirect handler for notebooks by name
- PR #3041: support non-modules in @require
- PR #2447: Stateful line transformers
- PR #3108: fix some O(N) and O(N^2) operations in parallel.map
- PR #2791: forward stdout from forked processes
- PR #3157: use Python 3-style for pretty-printed sets
- PR #3148: closes #3045, #3123 for tornado < version 3.0
- PR #3143: minor heading-link tweaks
- PR #3136: Strip useless ANSI escape codes in notebook
- PR #3126: Prevent errors when pressing arrow keys in an empty notebook
- PR #3135: quick dev installation instructions
- PR #2889: Push pandas dataframes to R magic
- PR #3068: Don't monkeypatch doctest during IPython startup.
- PR #3133: fix argparse version check
- PR #3102: set *spellcheck=false* in CodeCell inputarea
- PR #3064: add anchors to heading cells

- PR #3097: PyQt 4.10: use self._document = self.document()
- PR #3117: propagate automagic change to shell
- PR #3118: don't give up on weird os names
- PR #3115: Fix example
- PR #2640: fix quarantine/ipy_editors.py
- PR #3070: Add info make target that was missing in old Sphinx
- PR #3082: A few small patches to image handling
- PR #3078: fix regular expression for detecting links in stdout
- PR #3054: restore default behavior for automatic cluster size
- PR #3073: fix ipython usage text
- PR #3083: fix DisplayMagics.html docstring
- PR #3080: noted sub_channel being renamed to iopub_channel
- PR #3079: actually use IPKernelApp.kernel_class
- PR #3076: Improve notebook.js documentation
- PR #3063: add missing *%%html* magic
- PR #3075: check for SIGUSR1 before using it, closes #3074
- PR #3051: add width:100% to vbox for webkit / FF consistency
- PR #2999: increase registration timeout
- PR #2997: fix DictDB default size limit
- PR #3033: on resume, print server info again
- PR #3062: test double pyximport
- PR #3046: cast kernel cwd to bytes on Python 2 on Windows
- PR #3038: remove xml from notebook magic docstrings
- PR #3032: fix time format to international time format
- PR #3022: Fix test for Windows
- PR #3024: changed instances of 'outout' to 'output' in alt texts
- PR #3013: py3 workaround for reload in cythonmagic
- PR #2961: time magic: shorten unnecessary output on windows
- PR #2987: fix local files examples in markdown
- PR #2998: fix css in .output_area pre
- PR #3003: add $include /etc/inputrc to suggested ~/.inputrc
- PR #2957: Refactor qt import logic. Fixes #2955

- PR #2994: expanduser on %%file targets
- PR #2983: fix run-all (that-> this)
- PR #2964: fix count when testing composite error output
- PR #2967: shows entire session history when only startsess is given
- PR #2942: Move CM IPython theme out of codemirror folder
- PR #2929: Cleanup cell insertion
- PR #2933: Minordocupdate
- PR #2968: fix notebook deletion.
- PR #2966: Added assert msg to extract_hist_ranges()
- PR #2959: Add command to trim the history database.
- PR #2681: Don't enable pylab mode, when matplotlib is not importable
- PR #2901: Fix inputhook_wx on osx
- PR #2871: truncate potentially long CompositeErrors
- PR #2951: use istype on lists/tuples
- PR #2946: fix qtconsole history logic for end-of-line
- PR #2954: fix logic for append_javascript
- PR #2941: fix baseUrl
- PR #2903: Specify toggle value on cell line number
- PR #2911: display order in output area configurable
- PR #2897: Dont rely on BaseProjectUrl data in body tag
- PR #2894: Cm configurable
- PR #2927: next release will be 1.0
- PR #2932: Simplify using notebook static files from external code
- PR #2915: added small config section to notebook docs page
- PR #2924: safe_run_module: Silence SystemExit codes 0 and None.
- PR #2906: Unpatch/Monkey patch CM
- PR #2921: add menu item for undo delete cell
- PR #2917: Don't add logging handler if one already exists.
- PR #2910: Respect DB_IP and DB_PORT in mongodb tests
- PR #2926: Don't die if stderr/stdout do not support set_parent() #2925
- PR #2885: get monospace pager back
- PR #2876: fix celltoolbar layout on FF

- PR #2904: Skip remaining IPC test on Windows
- PR #2908: fix last remaining KernelApp reference
- PR #2905: fix a few remaining KernelApp/IPKernelApp changes
- PR #2900: Don’t assume test case for %time will finish in 0 time
- PR #2893: exclude fabfile from tests
- PR #2884: Correct import for kernelmanager on Windows
- PR #2882: Utils cleanup
- PR #2883: Don’t call ast.fix_missing_locations unless the AST could have been modified
- PR #2855: time(it) magic: Implement minutes/hour formatting and “%%time” cell magic
- PR #2874: Empty cell warnings
- PR #2819: tweak history prefix search (up/^p) in qtconsole
- PR #2868: Import performance
- PR #2877: minor css fixes
- PR #2880: update examples docs with kernel move
- PR #2878: Pass host environment on to kernel
- PR #2599: func_kw_complete for builtin and cython with embededsignature=True using docstring
- PR #2792: Add key “unique” to history_request protocol
- PR #2872: fix payload keys
- PR #2869: Fixing styling of toolbar selects on FF.
- PR #2708: Less css
- PR #2854: Move kernel code into IPython.kernel
- PR #2864: Fix %run -t -N<N> TypeError
- PR #2852: future pyzmq compatibility
- PR #2863: whatsnew/version0.9.txt: Fix ‘~./ipython’ -> ‘~/.ipython’ typo
- PR #2861: add missing KernelManager to ConsoleApp class list
- PR #2850: Consolidate host IP detection in utils.localinterfaces
- PR #2859: Correct docstring of ipython.py
- PR #2831: avoid string version comparisons in external.qt
- PR #2844: this should address the failure in #2732
- PR #2849: utils/data: Use list comprehension for uniq_stable()
- PR #2839: add jinja to install docs / setup.py
- PR #2841: Miscellaneous docs fixes

- PR #2811: Still more KernelManager cleanup
- PR #2820: add '=' to greedy completer delims
- PR #2818: log user tracebacks in the kernel (INFO-level)
- PR #2828: Clean up notebook Javascript
- PR #2829: avoid comparison error in dictdb hub history
- PR #2830: BUG: Opening parenthesis after non-callable raises ValueError
- PR #2718: try to fallback to pysqlite2.dbapi2 as sqlite3 in core.history
- PR #2816: in %edit, don't save "last_call" unless last call succeeded
- PR #2817: change ol format order
- PR #2537: Organize example notebooks
- PR #2815: update release/authors
- PR #2808: improve patience for slow Hub in client tests
- PR #2812: remove nonfunctional *-la* short arg in cython magic
- PR #2810: remove dead utils.upgradedir
- PR #1671: __future__ environments
- PR #2804: skip ipc tests on Windows
- PR #2789: Fixing styling issues with CellToolbar.
- PR #2805: fix KeyError creating ZMQStreams in notebook
- PR #2775: General cleanup of kernel manager code.
- PR #2340: Initial Code to reduce parallel.Client caching
- PR #2799: Exit code
- PR #2800: use *type(obj) is cls* as switch when canning
- PR #2801: Fix a breakpoint bug
- PR #2795: Remove outdated code from extensions.autoreload
- PR #2796: P3K: fix cookie parsing under Python 3.x (+ duplicate import is removed)
- PR #2724: In-process kernel support (take 3)
- PR #2687: [WIP] Metaui slideshow
- PR #2788: Chrome frame awareness
- PR #2649: Add version_request/reply messaging protocol
- PR #2753: add *%%px –local* for local execution
- PR #2783: Prefilter shouldn't touch execution_count
- PR #2333: UI For Metadata

- PR #2396: create a ipynbv3 json schema and a validator
- PR #2757: check for complete pyside presence before trying to import
- PR #2782: Allow the %run magic with '-b' to specify a file.
- PR #2778: P3K: fix DeprecationWarning under Python 3.x
- PR #2776: remove non-functional View.kill method
- PR #2755: can interactively defined classes
- PR #2774: Removing unused code in the notebook MappingKernelManager.
- PR #2773: Fixed minor typo causing AttributeError to be thrown.
- PR #2609: Add 'unique' option to history_request messaging protocol
- PR #2769: Allow shutdown when no engines are registered
- PR #2766: Define __file__ when we %edit a real file.
- PR #2476: allow %edit <variable> to work when interactively defined
- PR #2763: Reset readline delimiters after loading rmagic.
- PR #2460: Better handling of __*file*__ when running scripts.
- PR #2617: Fix for *units* argument. Adds a *res* argument.
- PR #2738: Unicode content crashes the pager (console)
- PR #2749: Tell Travis CI to test on Python 3.3 as well
- PR #2744: Don't show 'try %paste' message while using magics
- PR #2728: shift tab for tooltip
- PR #2741: Add note to *%cython* Black-Scholes example warning of missing erf.
- PR #2743: BUG: Octavemagic inline plots not working on Windows: Fixed
- PR #2740: Following #2737 this error is now a name error
- PR #2737: Rmagic: error message when moving an non-existant variable from python to R
- PR #2723: diverse fixes for project url
- PR #2731: %Rpush: Look for variables in the local scope first.
- PR #2544: Infinite loop when multiple debuggers have been attached.
- PR #2726: Add qthelp docs creation
- PR #2730: added blockquote CSS
- PR #2729: Fix Read the doc build, Again
- PR #2446: [alternate 2267] Offline mathjax
- PR #2716: remove unexisting headings level
- PR #2717: One liner to fix debugger printing stack traces when lines of context are larger than source.

- PR #2713: Doc bugfix: user_ns is not an attribute of Magic objects.
- PR #2690: Fix 'import '... completion for py3 & egg files.
- PR #2691: Document OpenMP in %%cython magic
- PR #2699: fix jinja2 rendering for password protected notebooks
- PR #2700: Skip notebook testing if jinja2 is not available.
- PR #2692: Add %%cython magics to generated documentation.
- PR #2685: Fix pretty print of types when __*module*__ is not available.
- PR #2686: Fix tox.ini
- PR #2604: Backslashes are misinterpreted as escape-sequences by the R-interpreter.
- PR #2689: fix error in doc (arg->kwarg) and pep-8
- PR #2683: for downloads, replaced window.open with window.location.assign
- PR #2659: small bugs in js are fixed
- PR #2363: Refactor notebook templates to use Jinja2
- PR #2662: qtconsole: wrap argument list in tooltip to match width of text body
- PR #2328: addition of classes to generate a link or list of links from files local to the IPython HTML notebook
- PR #2668: pylab_not_importable: Catch all exceptions, not just RuntimeErrors.
- PR #2663: Fix issue #2660: parsing of help and version arguments
- PR #2656: Fix irunner tests when $PYTHONSTARTUP is set
- PR #2312: Add bracket matching to code cells in notebook
- PR #2571: Start to document Javascript
- PR #2641: undefinied that -> this
- PR #2638: Fix %paste in Python 3 on Mac
- PR #2301: Ast transfomers
- PR #2616: Revamp API docs
- PR #2572: Make 'Paste Above' the default paste behavior.
- PR #2574: Fix #2244
- PR #2582: Fix displaying history when output cache is disabled.
- PR #2591: Fix for Issue #2584
- PR #2526: Don't kill paramiko tunnels when receiving ^C
- PR #2559: Add psource, pfile, pinfo2 commands to ipdb.
- PR #2546: use 4 Pythons to build 4 Windows installers

- PR #2561: Fix display of plain text containing multiple carriage returns before line feed
- PR #2549: Add a simple 'undo' for cell deletion.
- PR #2525: Add event to kernel execution/shell reply.
- PR #2554: Avoid stopping in ipdb until we reach the main script.
- PR #2404: Option to limit search result in history magic command
- PR #2294: inputhook_qt4: Use QEventLoop instead of starting up the QCoreApplication
- PR #2233: Refactored Drag and Drop Support in Qt Console
- PR #1747: switch between hsplit and vsplit paging (request for feedback)
- PR #2530: Adding time offsets to the video
- PR #2542: Allow starting IPython as *python -m IPython*.
- PR #2534: Do not unescape backslashes in Windows (shellglob)
- PR #2517: Improved MathJax, bug fixes
- PR #2511: trigger default remote_profile_dir when profile_dir is set
- PR #2491: color is supported in ironpython
- PR #2462: Track which extensions are loaded
- PR #2464: Locate URLs in text output and convert them to hyperlinks.
- PR #2490: add ZMQInteractiveShell to IPEngineApp class list
- PR #2498: Don't catch tab press when something selected
- PR #2527: Run All Above and Run All Below
- PR #2513: add GitHub uploads to release script
- PR #2529: Windows aware tests for shellglob
- PR #2478: Fix doctest_run_option_parser for Windows
- PR #2519: clear In[ ] prompt numbers again
- PR #2467: Clickable links
- PR #2500: Add *encoding* attribute to *OutStream* class.
- PR #2349: ENH: added StackExchange-style MathJax filtering
- PR #2503: Fix traceback handling of SyntaxErrors without line numbers.
- PR #2492: add missing 'qtconsole' extras_require
- PR #2480: Add deprecation warnings for sympyprinting
- PR #2334: Make the ipengine monitor the ipcontroller heartbeat and die if the ipcontroller goes down
- PR #2479: use new _winapi instead of removed _subprocess
- PR #2474: fix bootstrap name conflicts

- PR #2469: Treat __init__.pyc same as __init__.py in module_list
- PR #2165: Add -g option to %run to glob expand arguments
- PR #2468: Tell git to ignore __pycache__ directories.
- PR #2421: Some notebook tweaks.
- PR #2291: Remove old plugin system
- PR #2127: Ability to build toolbar in JS
- PR #2445: changes for ironpython
- PR #2420: Pass ipython_dir to __init__() method of TerminalInteractiveShell's superclass.
- PR #2432: Revert #1831, the __*file*__ injection in safe_execfile / safe_execfile_ipy.
- PR #2216: Autochange highlight with cell magics
- PR #1946: Add image message handler in ZMQTerminalInteractiveShell
- PR #2424: skip find_cmd when setting up script magics
- PR #2389: Catch sqlite DatabaseErrors in more places when reading the history database
- PR #2395: Don't catch ImportError when trying to unpack module functions
- PR #1868: enable IPC transport for kernels
- PR #2437: don't let log cleanup prevent engine start
- PR #2441: *sys.maxsize* is the maximum length of a container.
- PR #2442: allow iptest to be interrupted
- PR #2240: fix message built for engine dying during task
- PR #2369: Block until kernel termination after sending a kill signal
- PR #2439: Py3k: Octal (0777 -> 0o777)
- PR #2326: Detachable pager in notebook.
- PR #2377: Fix installation of man pages in Python 3
- PR #2407: add IPython version to message headers
- PR #2408: Fix Issue #2366
- PR #2405: clarify TaskScheduler.hwm doc
- PR #2399: IndentationError display
- PR #2400: Add scroll_to_cell(cell_number) to the notebook
- PR #2401: unmock read-the-docs modules
- PR #2311: always perform requested trait assignments
- PR #2393: New option *n* to limit history search hits
- PR #2386: Adapt inline backend to changes in matplotlib

- PR #2392: Remove suspicious double quote
- PR #2387: Added -L library search path to cythonmagic cell magic
- PR #2370: qtconsole: Create a prompt newline by inserting a new block (w/o formatting)
- PR #1715: Fix for #1688, traceback-unicode issue
- PR #2378: use Singleton.instance() for embed() instead of manual global
- PR #2373: fix missing imports in core.interactiveshell
- PR #2368: remove notification widget leftover
- PR #2327: Parallel: Support get/set of nested objects in view (e.g. dv['a.b'])
- PR #2362: Clean up ProgressBar class in example notebook
- PR #2346: Extra xterm identification in set_term_title
- PR #2352: Notebook: Store the username in a cookie whose name is unique.
- PR #2358: add backport_pr to tools
- PR #2365: fix names of notebooks for download/save
- PR #2364: make clients use 'location' properly (fixes #2361)
- PR #2354: Refactor notebook templates to use Jinja2
- PR #2339: add bash completion example
- PR #2345: Remove references to 'version' no longer in argparse. Github issue #2343.
- PR #2347: adjust division error message checking to account for Python 3
- PR #2305: RemoteError._render_traceback_ calls self.render_traceback
- PR #2338: Normalize line endings for ipexec_validate, fix for #2315.
- PR #2192: Introduce Notification Area
- PR #2329: Better error messages for common magic commands.
- PR #2337: ENH: added StackExchange-style MathJax filtering
- PR #2331: update css for qtconsole in doc
- PR #2317: adding cluster_id to parallel.Client.__init__
- PR #2130: Add -l option to %R magic to allow passing in of local namespace
- PR #2196: Fix for bad command line argument to latex
- PR #2300: bug fix: was crashing when sqlite3 is not installed
- PR #2184: Expose store_history to execute_request messages.
- PR #2308: Add welcome_message option to enable_pylab
- PR #2302: Fix variable expansion on 'self'
- PR #2299: Remove code from prefilter that duplicates functionality in inputsplitter

- PR #2295: allow pip install from github repository directly
- PR #2280: fix SSH passwordless check for OpenSSH
- PR #2290: nbmanager
- PR #2288: s/assertEquals/assertEqual (again)
- PR #2287: Removed outdated dev docs.
- PR #2218: Use redirect for new notebooks
- PR #2277: nb: up/down arrow keys move to begin/end of line at top/bottom of cell
- PR #2045: Refactoring notebook managers and adding Azure backed storage.
- PR #2271: use display instead of send_figure in inline backend hooks
- PR #2278: allow disabling SQLite history
- PR #2225: Add "–annotate" option to *%%cython* magic.
- PR #2246: serialize individual args/kwargs rather than the containers
- PR #2274: CLN: Use name to id mapping of notebooks instead of searching.
- PR #2270: SSHLauncher tweaks
- PR #2269: add missing location when disambiguating controller IP
- PR #2263: Allow docs to build on http://readthedocs.org/
- PR #2256: Adding data publication example notebook.
- PR #2255: better flush iopub with AsyncResults
- PR #2261: Fix: longest_substr([]) -> ''
- PR #2260: fix mpr again
- PR #2242: Document globbing in *%history -g <pattern>*.
- PR #2250: fix html in notebook example
- PR #2245: Fix regression in embed() from pull-request #2096.
- PR #2248: track sha of master in test_pr messages
- PR #2238: Fast tests
- PR #2211: add data publication message
- PR #2236: minor test_pr tweaks
- PR #2231: Improve Image format validation and add html width,height
- PR #2232: Reapply monkeypatch to inspect.findsource()
- PR #2235: remove spurious print statement from setupbase.py
- PR #2222: adjust how canning deals with import strings
- PR #2224: fix css typo

- PR #2223: Custom tracebacks
- PR #2214: use KernelApp.exec_lines/files in IPEngineApp
- PR #2199: Wrap JS published by %%javascript in try/catch
- PR #2212: catch errors in markdown javascript
- PR #2190: Update code mirror 2.22 to 2.32
- PR #2200: documentation build broken in bb429da5b
- PR #2194: clean nan/inf in json_clean
- PR #2198: fix mpr for earlier git version
- PR #2175: add FileFindHandler for Notebook static files
- PR #1990: can func_defaults
- PR #2069: start improving serialization in parallel code
- PR #2202: Create a unique & temporary IPYTHONDIR for each testing group.
- PR #2204: Work around lack of os.kill in win32.
- PR #2148: win32 iptest: Use subprocess.Popen() instead of os.system().
- PR #2179: Pylab switch
- PR #2124: Add an API for registering magic aliases.
- PR #2169: ipdb: pdef, pdoc, pinfo magics all broken
- PR #2174: Ensure consistent indentation in *%magic*.
- PR #1930: add size-limiting to the DictDB backend
- PR #2189: Fix IPython.lib.latextools for Python 3
- PR #2186: removed references to h5py dependence in octave magic documentation
- PR #2183: Include the kernel object in the event object passed to kernel events
- PR #2185: added test for %store, fixed storemagic
- PR #2138: Use breqn.sty in dvipng backend if possible
- PR #2182: handle undefined param in notebooklist
- PR #1831: fix #1814 set __file__ when running .ipy files
- PR #2051: Add a metadata attribute to messages
- PR #1471: simplify IPython.parallel connections and enable Controller Resume
- PR #2181: add %%javascript, %%svg, and %%latex display magics
- PR #2116: different images in 00_notebook-tour
- PR #2092: %prun: Restore *stats.stream* after running *print_stream*.
- PR #2159: show message on notebook list if server is unreachable

- PR #2176: fix git mpr
- PR #2152: [qtconsole] Namespace not empty at startup
- PR #2177: remove numpy install from travis/tox scripts
- PR #2090: New keybinding for code cell execution + cell insertion
- PR #2160: Updating the parallel options pricing example
- PR #2168: expand line in cell magics
- PR #2170: Fix tab completion with IPython.embed_kernel().
- PR #2096: embed(): Default to the future compiler flags of the calling frame.
- PR #2163: fix 'remote_profie_dir' typo in SSH launchers
- PR #2158: [2to3 compat] Tuple params in func defs
- PR #2089: Fix unittest DeprecationWarnings
- PR #2142: Refactor test_pr.py
- PR #2140: 2to3: Apply *has_key* fixer.
- PR #2131: Add option append (-a) to %save
- PR #2117: use explicit url in notebook example
- PR #2133: Tell git that *.py files contain Python code, for use in word-diffs.
- PR #2134: Apply 2to3 *next* fix.
- PR #2126: ipcluster broken with any batch launcher (PBS/LSF/SGE)
- PR #2104: Windows make file for Sphinx documentation
- PR #2074: Make BG color of inline plot configurable
- PR #2123: BUG: Look up the *_repr_pretty_* method on the class within the MRO rath...
- PR #2100: [in progress] python 2 and 3 compatibility without 2to3, second try
- PR #2128: open notebook copy in different tabs
- PR #2073: allows password and prefix for notebook
- PR #1993: Print View
- PR #2086: re-aliad %ed to %edit in qtconsole
- PR #2110: Fixes and improvements to the input splitter
- PR #2101: fix completer deletting newline
- PR #2102: Fix logging on interactive shell.
- PR #2088: Fix (some) Python 3.2 ResourceWarnings
- PR #2064: conform to pep 3110
- PR #2076: Skip notebook 'static' dir in test suite.

- PR #2063: Remove umlauts so py3 installations on LANG=C systems succeed.
- PR #2068: record sysinfo in sdist
- PR #2067: update tools/release_windows.py
- PR #2065: Fix parentheses typo
- PR #2062: Remove duplicates and auto-generated files from repo.
- PR #2061: use explicit tuple in exception
- PR #2060: change minus to - or (hy in manpages

Issues (691):

- #3940: Install process documentation overhaul
- #3946: The PDF option for *–post* should work with lowercase
- #3957: Notebook help page broken in Firefox
- #3894: nbconvert test failure
- #3887: 1.0.0a1 shows blank screen in both firefox and chrome (windows 7)
- #3703: *nbconvert*: Output options – names and documentataion
- #3931: Tab completion not working during debugging in the notebook
- #3936: Ipcluster plugin is not working with Ipython 1.0dev
- #3941: IPython Notebook kernel crash on Win7x64
- #3926: Ending Notebook renaming dialog with return creates new-line
- #3932: Incorrect empty docstring
- #3928: Passing variables to script from the workspace
- #3774: Notebooks with spaces in their names breaks nbconvert latex graphics
- #3916: tornado needs its own check
- #3915: Link to Parallel examples "found on GitHub" broken in docs
- #3895: Keyboard shortcuts box in notebook doesn't fit the screen
- #3912: IPython.utils fails automated test for RC1 1.0.0
- #3636: Code cell missing highlight on load
- #3897: under Windows, "ipython3 nbconvert "C:/blabla/first_try.ipynb" –to latex –post PDF" POST processing action fails because of a bad parameter
- #3900: python3 install syntax errors (OS X 10.8.4)
- #3899: nbconvert to latex fails on notebooks with spaces in file name
- #3881: Temporary Working Directory Test Fails
- #2750: A way to freeze code cells in the notebook

- #3893: Resize Local Image Files in Notebook doesn't work
- #3823: nbconvert on windows: tex and paths
- #3885: under Windows, "ipython3 nbconvert "C:/blabla/first_try.ipynb" –to latex" write "" instead of "/" to reference file path in the .tex file
- #3889: test_qt fails due to assertion error 'qt4' != 'qt'
- #3890: double post, disregard this issue
- #3689: nbconvert, remaining tests
- #3874: Up/Down keys don't work to "Search previous command history" (besides Ctrl-p/Ctrl-n)
- #3853: CodeMirror locks up in the notebook
- #3862: can only connect to an ipcluster started with v1.0.0-dev (master branch) using an older ipython (v0.13.2), but cannot connect using ipython (v1.0.0-dev)
- #3869: custom css not working.
- #2960: Keyboard shortcuts
- #3795: ipcontroller process goes to 100% CPU, ignores connection requests
- #3553: Ipython and pylab crashes in windows and canopy
- #3837: Cannot set custom mathjax url, crash notebook server.
- #3808: "Naming" releases ?
- #2431: TypeError: must be string without null bytes, not str
- #3856: *?* at end of comment causes line to execute
- #3731: nbconvert: add logging for the different steps of nbconvert
- #3835: Markdown cells do not render correctly when mathjax is disabled
- #3843: nbconvert to rst: leftover "In[ ]"
- #3799: nbconvert: Ability to specify name of output file
- #3726: Document when IPython.start_ipython() should be used versus IPython.embed()
- #3778: Add no more readonly view in what's new
- #3754: No Print View in Notebook in 1.0dev
- #3798: IPython 0.12.1 Crashes on autocompleting sqlalchemy.func.row_number properties
- #3811: Opening notebook directly from the command line with multi-directory support installed
- #3775: Annoying behavior when clicking on cell after execution (Ctrl+Enter)
- #3809: Possible to add some bpython features?
- #3810: Printing the contents of an image file messes up shell text
- #3702: *nbconvert*: Default help message should be that of –help
- #3735: Nbconvert 1.0.0a1 does not take into account the pdf extensions in graphs

- #3719: Bad strftime format, for windows, in nbconvert exporter
- #3786: Zmq errors appearing with *Ctrl-C* in console/qtconsole
- #3019: disappearing scrollbar on tooltip in Chrome 24 on Ubuntu 12.04
- #3785: ipdb completely broken in Qt console
- #3796: Document the meaning of milestone/issues-tags for users.
- #3788: Do not auto show tooltip if docstring empty.
- #1366: [Web page] No link to front page from documentation
- #3739: nbconvert (to slideshow) misses some of the math in markdown cells
- #3768: increase and make timeout configurable in console completion.
- #3724: ipcluster only running on one cpu
- #1592: better message for unsupported nbformat
- #2049: Can not stop "ipython kernel" on windows
- #3757: Need direct entry point to given notebook
- #3745: ImportError: cannot import name check_linecache_ipython
- #3701: *nbconvert*: Final output file should be in same directory as input file
- #3738: history -o works but history with -n produces identical results
- #3740: error when attempting to run 'make' in docs directory
- #3737: ipython nbconvert crashes with ValueError: Invalid format string.
- #3730: nbconvert: unhelpful error when pandoc isn't installed
- #3718: markdown cell cursor misaligned in notebook
- #3710: mutiple input fields for %debug in the notebook after resetting the kernel
- #3713: PyCharm has problems with IPython working inside PyPy created by virtualenv
- #3712: Code completion: Complete on dictionary keys
- #3680: –pylab and –matplotlib flag
- #3698: nbconvert: Unicode error with minus sign
- #3693: nbconvert does not process SVGs into PDFs
- #3688: nbconvert, figures not extracting with Python 3.x
- #3542: note new dependencies in docs / setup.py
- #2556: [pagedown] do not target_blank anchor link
- #3684: bad message when %pylab fails due import *other* than matplotlib
- #3682: ipython notebook pylab inline import_all=False
- #3596: MathjaxUtils race condition?

- #1540: Comment/uncomment selection in notebook
- #2702: frozen setup: permission denied for default ipython_dir
- #3672: allow_none on Number-like traits.
- #2411: add CONTRIBUTING.md
- #481: IPython terminal issue with Qt4Agg on XP SP3
- #2664: How to preserve user variables from import clashing?
- #3436: enable_pylab(import_all=False) still imports np
- #2630: lib.pylabtools.figsize : NameError when using Qt4Agg backend and %pylab magic.
- #3154: Notebook: no event triggered when a Cell is created
- #3579: Nbconvert: SVG are not transformed to PDF anymore
- #3604: MathJax rendering problem in *%%latex* cell
- #3668: AttributeError: 'BlockingKernelClient' object has no attribute 'started_channels'
- #3245: SyntaxError: encoding declaration in Unicode string
- #3639: %pylab inline in IPYTHON notebook throws "RuntimeError: Cannot activate multiple GUI eventloops"
- #3663: frontend deprecation warnings
- #3661: run -m not behaving like python -m
- #3597: re-do PR #3531 - allow markdown in Header cell
- #3053: Markdown in header cells is not rendered
- #3655: IPython finding its way into pasted strings.
- #3620: uncaught errors in HTML output
- #3646: get_dict() error
- #3004: *%load_ext rmagic* fails when legacy ipy_user_conf.py is installed (in ipython 0.13.1 / OSX 10.8)
- #3638: setp() issue in ipython notebook with figure references
- #3634: nbconvert reveal to pdf conversion ignores styling, prints only a single page.
- #1307: Remove pyreadline workarounds, we now require pyreadline >= 1.7.1
- #3316: find_cmd test failure on Windows
- #3494: input() in notebook doesn't work in Python 3
- #3427: Deprecate *$* as mathjax delimiter
- #3625: Pager does not open from button
- #3149: Miscellaneous small nbconvert feedback
- #3617: 256 color escapes support

- #3609: %pylab inline blows up for single process ipython
- #2934: Publish the Interactive MPI Demo Notebook
- #3614: ansi escapes broken in master (ls –color)
- #3610: If you don’t have markdown, python setup.py install says no pygments
- #3547: %run modules clobber each other
- #3602: import_item fails when one tries to use DottedObjectName instead of a string
- #3563: Duplicate tab completions in the notebook
- #3599: Problems trying to run IPython on python3 without installing...
- #2937: too long completion in notebook
- #3479: Write empty name for the notebooks
- #3505: nbconvert: Failure in specifying user filter
- #1537: think a bit about namespaces
- #3124: Long multiline strings in Notebook
- #3464: run -d message unclear
- #2706: IPython 0.13.1 ignoring $PYTHONSTARTUP
- #3587: LaTeX escaping bug in nbconvert when exporting to HTML
- #3213: Long running notebook died with a coredump
- #3580: Running ipython with pypy on windows
- #3573: custom.js not working
- #3544: IPython.lib test failure on Windows
- #3352: Install Sphinx extensions
- #2971: [notebook]user needs to press ctrl-c twice to stop notebook server should be put into terminal window
- #2413: ipython3 qtconsole fails to install: ipython 0.13 has no such extra feature ‘qtconsole’
- #2618: documentation is incorrect for install process
- #2595: mac 10.8 qtconsole export history
- #2586: cannot store aliases
- #2714: ipython qtconsole print unittest messages in console instead his own window.
- #2669: cython magic failing to work with openmp.
- #3256: Vagrant pandas instance of iPython Notebook does not respect additional plotting arguments
- #3010: cython magic fail if cache dir is deleted while in session
- #2044: prune unused names from parallel.error

- #1145: Online help utility broken in QtConsole
- #3439: Markdown links no longer open in new window (with change from pagedown to marked)
- #3476: _margv for macros seems to be missing
- #3499: Add reveal.js library (version 2.4.0) inside IPython
- #2771: Wiki Migration to GitHub
- #2887: ipcontroller purging some engines during connect
- #626: Enable Resuming Controller
- #2824: Kernel restarting after message "Kernel XXXX failed to respond to heartbeat"
- #2823: %%cython magic gives ImportError: dlopen(long_file_name.so, 2): image not found
- #2891: In IPython for Python 3, system site-packages comes before user site-packages
- #2928: Add magic "watch" function (example)
- #2931: Problem rendering pandas dataframe in Firefox for Windows
- #2939: [notebook] Figure legend not shown in inline backend if ouside the box of the axes
- #2972: [notebook] in Markdown mode, press Enter key at the end of <some http link>, the next line is indented unexpectly
- #3069: Instructions for installing IPython notebook on Windows
- #3444: Encoding problem: cannot use if user's name is not ascii?
- #3335: Reenable bracket matching
- #3386: Magic %paste not working in Python 3.3.2. TypeError: Type str doesn't support the buffer API
- #3543: Exception shutting down kernel from notebook dashboard (0.13.1)
- #3549: Codecell size changes with selection
- #3445: Adding newlines in %%latex cell
- #3237: [notebook] Can't close a notebook without errors
- #2916: colon invokes auto(un)indent in markdown cells
- #2167: Indent and dedent in htmlnotebook
- #3545: Notebook save button icon not clear
- #3534: nbconvert incompatible with Windows?
- #3489: Update example notebook that raw_input is allowed
- #3396: Notebook checkpoint time is displayed an hour out
- #3261: Empty revert to checkpoint menu if no checkpoint...
- #2984: "print" magic does not work in Python 3
- #3524: Issues with pyzmq and ipython on EPD update

- #2434: %store magic not auto-restoring
- #2720: base_url and static path
- #2234: Update various low resolution graphics for retina displays
- #2842: Remember passwords for pw-protected notebooks
- #3244: qtconsole: ValueError('close_fds is not supported on Windows platforms if you redirect stdin/stdout/stderr',)
- #2215: AsyncResult.wait(0) can hang waiting for the client to get results?
- #2268: provide mean to retrieve static data path
- #1905: Expose UI for worksheets within each notebook
- #2380: Qt inputhook prevents modal dialog boxes from displaying
- #3185: prettify on double //
- #2821: Test failure: IPython.parallel.tests.test_client.test_resubmit_header
- #2475: [Notebook] Line is deindented when typing eg a colon in markdown mode
- #2470: Do not destroy valid notebooks
- #860: Allow the standalone export of a notebook to HTML
- #2652: notebook with qt backend crashes at save image location popup
- #1587: Improve kernel restarting in the notebook
- #2710: Saving a plot in Mac OS X backend crashes IPython
- #2596: notebook "Last saved:" is misleading on file opening.
- #2671: TypeError :NoneType when executed "ipython qtconsole" in windows console
- #2703: Notebook scrolling breaks after pager is shown
- #2803: KernelManager and KernelClient should be two separate objects
- #2693: TerminalIPythonApp configuration fails without ipython_config.py
- #2531: IPython 0.13.1 python 2 32-bit installer includes 64-bit ipython*.exe launchers in the scripts folder
- #2520: Control-C kills port forwarding
- #2279: Setting __*file*__ to None breaks Mayavi import
- #2161: When logged into notebook, long titles are incorrectly positioned
- #1292: Notebook, Print view should not be editable...
- #1731: test parallel launchers
- #3227: Improve documentation of ipcontroller and possible BUG
- #2896: IPController very unstable
- #3517: documentation build broken in head

- #3522: UnicodeDecodeError: ‘ascii’ codec can’t decode byte on Pycharm on Windows
- #3448: Please include MathJax fonts with IPython Notebook
- #3519: IPython Parallel map mysteriously turns pandas Series into numpy ndarray
- #3345: IPython embedded shells ask if I want to exit, but I set confirm_exit = False
- #3509: IPython won’t close without asking “Are you sure?” in Firefox
- #3471: Notebook jinja2/markupsafe depedencies in manual
- #3502: Notebook broken in master
- #3302: autoreload does not work in ipython 0.13.x, python 3.3
- #3475: no warning when leaving/closing notebook on master without saved changes
- #3490: No obvious feedback when kernel crashes
- #1912: Move all autoreload tests to their own group
- #2577: sh.py and ipython for python 3.3
- #3467: %magic doesn’t work
- #3501: Editing markdown cells that wrap has off-by-one errors in cursor positioning
- #3492: IPython for Python3
- #3474: unexpected keyword argument to remove_kernel
- #2283: TypeError when using ‘?’ after a string in a %logstart session
- #2787: rmagic and pandas DataFrame
- #2605: Ellipsis literal triggers AttributeError
- #1179: Test unicode source in pinfo
- #2055: drop Python 3.1 support
- #2293: IPEP 2: Input transformations
- #2790: %paste and %cpaste not removing ”...” lines
- #3480: Testing fails because iptest.py cannot be found
- #2580: will not run within PIL build directory
- #2797: RMagic, Dataframe Conversion Problem
- #2838: Empty lines disappear from triple-quoted literals.
- #3050: Broken link on IPython.core.display page
- #3473: Config not passed down to subcommands
- #3462: Setting log_format in config file results in error (and no format changes)
- #3311: Notebook (occasionally) not working on windows (Sophos AV)
- #3461: Cursor positioning off by a character in auto-wrapped lines

- #3454: _repr_html_ error
- #3457: Space in long Paragraph Markdown cell with Chinese or Japanese
- #3447: Run Cell Does not Work
- #1373: Last lines in long cells are hidden
- #1504: Revisit serialization in IPython.parallel
- #1459: Can't connect to 2 HTTPS notebook servers on the same host
- #678: Input prompt stripping broken with multiline data structures
- #3001: IPython.notebook.dirty flag is not set when a cell has unsaved changes
- #3077: Multiprocessing semantics in parallel.view.map
- #3056: links across notebooks
- #3120: Tornado 3.0
- #3156: update pretty to use Python 3 style for sets
- #3197: Can't escape multiple dollar signs in a markdown cell
- #3309: *Image()* signature/doc improvements
- #3415: Bug in IPython/external/path/__init__.py
- #3446: Feature suggestion: Download matplotlib figure to client browser
- #3295: autoexported notebooks: only export explicitly marked cells
- #3442: Notebook: Summary table extracted from markdown headers
- #3438: Zooming notebook in chrome is broken in master
- #1378: Implement autosave in notebook
- #3437: Highlighting matching parentheses
- #3435: module search segfault
- #3424: ipcluster –version
- #3434: 0.13.2 Ipython/genutils.py doesn't exist
- #3426: Feature request: Save by cell and not by line #: IPython %save magic
- #3412: Non Responsive Kernel: Running a Django development server from an IPython Notebook
- #3408: Save cell toolbar and slide type metadata in notebooks
- #3246: %paste regression with blank lines
- #3404: Weird error with $variable and grep in command line magic (!command)
- #3405: Key auto-completion in dictionaries?
- #3259: Codemirror linenumber css broken
- #3397: Vertical text misalignment in Markdown cells

- #3391: Revert #3358 once fix integrated into CM
- #3360: Error 500 while saving IPython notebook
- #3375: Frequent Safari/Webkit crashes
- #3365: zmq frontend
- #2654: User_expression issues
- #3389: Store history as plain text
- #3388: Ipython parallel: open TCP connection created for each result returned from engine
- #3385: setup.py failure on Python 3
- #3376: Setting __*module*__ to None breaks pretty printing
- #3374: ipython qtconsole does not display the prompt on OSX
- #3380: simple call to kernel
- #3379: TaskRecord key 'started' not set
- #3241: notebook conection time out
- #3334: magic interpreter interpretes non magic commands?
- #3326: python3.3: Type error when launching SGE cluster in IPython notebook
- #3349: pip3 doesn't run 2to3?
- #3347: Longlist support in ipdb
- #3343: Make pip install / easy_install faster
- #3337: git submodules broke nightly PPA builds
- #3206: Copy/Paste Regression in QtConsole
- #3329: Buggy linewrap in Mac OSX Terminal (Mountain Lion)
- #3327: Qt version check broken
- #3303: parallel tasks never finish under heavy load
- #1381: '\' for equation continuations require an extra '' in markdown cells
- #3314: Error launching iPython
- #3306: Test failure when running on a Vagrant VM
- #3280: IPython.utils.process.getoutput returns stderr
- #3299: variables named _ or __ exhibit incorrect behavior
- #3196: add an "x" or similar to htmlnotebook pager
- #3293: Several 404 errors for js files Firefox
- #3292: syntax highlighting in chrome on OSX 10.8.3
- #3288: Latest dev version hangs on page load

- #3283: ipython dev retains directory information after directory change
- #3279: custom.css is not overridden in the dev IPython (1.0)
- #2727: %run -m doesn't support relative imports
- #3268: GFM triple backquote and unknown language
- #3273: Suppressing all plot related outputs
- #3272: Backspace while completing load previous page
- #3260: Js error in savewidget
- #3247: scrollbar in notebook when not needed?
- #3243: notebook: option to view json source from browser
- #3265: 404 errors when running IPython 1.0dev
- #3257: setup.py not finding submodules
- #3253: Incorrect Qt and PySide version comparison
- #3248: Cell magics broken in Qt console
- #3012: Problems with the less based style.min.css
- #2390: Image width/height don't work in embedded images
- #3236: cannot set TerminalIPythonApp.log_format
- #3214: notebook kernel dies if started with invalid parameter
- #2980: Remove HTMLCell ?
- #3128: qtconsole hangs on importing pylab (using X forwarding)
- #3198: Hitting recursive depth causing all notebook pages to hang
- #3218: race conditions in profile directory creation
- #3177: OverflowError execption in handlers.py
- #2563: core.profiledir.check_startup_dir() doesn't work inside py2exe'd installation
- #3207: [Feature] folders for ipython notebook dashboard
- #3178: cell magics do not work with empty lines after #2447
- #3204: Default plot() colors unsuitable for red-green colorblind users
- #1789: :n/*foo turns into :n*(foo) in triple-quoted strings.
- #3202: File cell magic fails with blank lines
- #3199: %%cython -a stopped working?
- #2688: obsolete imports in import autocompletion
- #3192: Python2, Unhandled exception, __builtin__.True = False
- #3179: script magic error message loop

- #3009: use XDG_CACHE_HOME for cython objects
- #3059: Bugs in 00_notebook_tour example.
- #3104: Integrate a javascript file manager into the notebook front end
- #3176: Particular equation not rendering (notebook)
- #1133: [notebook] readonly and upload files/UI
- #2975: [notebook] python file and cell toolbar
- #3017: SciPy.weave broken in IPython notebook/ qtconsole
- #3161: paste macro not reading spaces correctly
- #2835: %paste not working on WinXpSP3/ipython-0.13.1.py2-win32-PROPER.exe/python27
- #2628: Make transformers work for lines following decorators
- #2612: Multiline String containing ":n?foon" confuses interpreter to replace ?foo with get_ipython().magic(u'pinfo foo')
- #2539: Request: Enable cell magics inside of .ipy scripts
- #2507: Multiline string does not work (includes ...) with doctest type input in IPython notebook
- #2164: Request: Line breaks in line magic command
- #3106: poor parallel performance with many jobs
- #2438: print inside multiprocessing crashes Ipython kernel
- #3155: Bad md5 hash for package 0.13.2
- #3045: [Notebook] Ipython Kernel does not start if disconnected from internet(/network?)
- #3146: Using celery in python 3.3
- #3145: The notebook viewer is down
- #2385: grep –color not working well with notebook
- #3131: Quickly install from source in a clean virtualenv?
- #3139: Rolling log for ipython
- #3127: notebook with pylab=inline appears to call figure.draw twice
- #3129: Walking up and down the call stack
- #3123: Notebook crashed if unplugged ethernet cable
- #3121: NB should use normalize.css? was #3049
- #3087: Disable spellchecking in notebook
- #3084: ipython pyqt 4.10 incompatibilty, QTextBlockUserData
- #3113: Fails to install under Jython 2.7 beta
- #3110: Render of h4 headers is not correct in notebook (error in renderedhtml.css)

- #3109: BUG: read_csv: dtype={'id' : np.str}: Datatype not understood
- #3107: Autocompletion of object attributes in arrays
- #3103: Reset locale setting in qtconsole
- #3090: python3.3 Entry Point not found
- #3081: UnicodeDecodeError when using Image(data="some.jpeg")
- #2834: url regexp only finds one link
- #3091: qtconsole breaks doctest.testmod() in Python 3.3
- #3074: SIGUSR1 not available on Windows
- #2996: registration::purging stalled registration high occurrence in small clusters
- #3065: diff-ability of notebooks
- #3067: Crash with pygit2
- #3061: Bug handling Ellipsis
- #3049: NB css inconsistent behavior between ff and webkit
- #3039: unicode errors when opening a new notebook
- #3048: Installning ipython qtConsole should be easyer att Windows
- #3042: Profile creation fails on 0.13.2 branch
- #3035: docstring typo/inconsistency: mention of an xml notebook format?
- #3031: HDF5 library segfault (possibly due to mismatching headers?)
- #2991: In notebook importing sympy closes ipython kernel
- #3027: f.__globals__ causes an error in Python 3.3
- #3020: Failing test test_interactiveshell.TestAstTransform on Windows
- #3023: alt text for "click to expand output" has typo in alt text
- #2963: %history to print all input history of a previous session when line range is omitted
- #3018: IPython installed within virtualenv. WARNING "Please install IPython inside the virtualtenv"
- #2484: Completion in Emacs *Python* buffer causes prompt to be increased.
- #3014: Ctrl-C finishes notebook immediately
- #3007: cython_pyximport reload broken in python3
- #2955: Incompatible Qt imports when running inprocess_qtconsole
- #3006: [IPython 0.13.1] The check of PyQt version is wrong
- #3005: Renaming a notebook to an existing notebook name overwrites the other file
- #2940: Abort trap in IPython Notebook after installing matplotlib
- #3000: issue #3000

- #2995: ipython_directive.py fails on multiline when prompt number < 100
- #2993: File magic (%%file) does not work with paths beginning with tilde (e.g., ~/anaconda/stuff.txt)
- #2992: Cell-based input for console and qt frontends?
- #2425: Liaise with Spyder devs to integrate newer IPython
- #2986: requesting help in a loop can damage a notebook
- #2978: v1.0-dev build errors on Arch with Python 3.
- #2557: [refactor] Insert_cell_at_index()
- #2969: ipython command does not work in terminal
- #2762: OSX wxPython (osx_cocoa, 64bit) command "%gui wx" blocks the interpreter
- #2956: Silent importing of submodules differs from standard Python3.2 interpreter's behavior
- #2943: Up arrow key history search gets stuck in QTConsole
- #2953: using 'nonlocal' declaration in global scope causes ipython3 crash
- #2952: qtconsole ignores exec_lines
- #2949: ipython crashes due to atexit()
- #2947: From rmagic to an R console
- #2938: docstring pane not showing in notebook
- #2936: Tornado assumes invalid signature for parse_qs on Python 3.1
- #2935: unable to find python after easy_install / pip install
- #2920: Add undo-cell deletion menu
- #2914: BUG:saving a modified .py file after loading a module kills the kernel
- #2925: BUG: kernel dies if user sets sys.stderr or sys.stdout to a file object
- #2909: LaTeX sometimes fails to render in markdown cells with some curly bracket + underscore combinations
- #2898: Skip ipc tests on Windows
- #2902: ActiveState attempt to build ipython 0.12.1 for python 3.2.2 for Mac OS failed
- #2899: Test failure in IPython.core.tests.test_magic.test_time
- #2890: Test failure when fabric not installed
- #2892: IPython tab completion bug for paths
- #1340: Allow input cells to be collapsed
- #2881: ? command in notebook does not show help in Safari
- #2751: %%timeit should use minutes to format running time in long running cells
- #2879: When importing a module with a wrong name, ipython crashes

- #2862: %%timeit should warn of empty contents
- #2485: History navigation breaks in qtconsole
- #2785: gevent input hook
- #2843: Sliently running code in clipboard (with paste, cpaste and variants)
- #2784: %run -t -N<N> error
- #2732: Test failure with FileLinks class on Windows
- #2860: ipython help notebook -> KeyError: ‘KernelManager’
- #2858: Where is the installed *ipython* script?
- #2856: Edit code entered from ipython in external editor
- #2722: IPC transport option not taking effect ?
- #2473: Better error messages in ipengine/ipcontroller
- #2836: Cannot send builtin module definitions to IP engines
- #2833: Any reason not to use super() ?
- #2781: Cannot interrupt infinite loops in the notebook
- #2150: clippath_demo.py in matplotlib example does not work with inline backend
- #2634: Numbered list in notebook markdown cell renders with Roman numerals instead of numbers
- #2230: IPython crashing during startup with “AttributeError: ‘NoneType’ object has no attribute ‘rstrip’”
- #2483: nbviewer bug? with multi-file gists
- #2466: mistyping *ed -p* breaks *ed -p*
- #2477: Glob expansion tests fail on Windows
- #2622: doc issue: notebooks that ship with Ipython .13 are written for python 2.x
- #2626: Add “Cell -> Run All Keep Going” for notebooks
- #1223: Show last modification date of each notebook
- #2621: user request: put link to example notebooks in Dashboard
- #2564: grid blanks plots in ipython pylab inline mode (interactive)
- #2532: Django shell (IPython) gives NameError on dict comprehensions
- #2188: ipython crashes on ctrl-c
- #2391: Request: nbformat API to load/save without changing version
- #2355: Restart kernel message even though kernel is perfectly alive
- #2306: Garbled input text after reverse search on Mac OS X
- #2297: ipdb with separate kernel/client pushing stdout to kernel process only

- #2180: Have [kernel busy] overridden only by [kernel idle]
- #1188: Pylab with OSX backend keyboard focus issue and hang
- #2107: test_octavemagic.py[everything] fails
- #1212: Better understand/document browser compatibility
- #1585: Refactor notebook templates to use Jinja2 and make each page a separate directory
- #1443: xticks scaling factor partially obscured with qtconsole and inline plotting
- #1209: can't make %result work as in doc.
- #1200: IPython 0.12 Windows install fails on Vista
- #1127: Interactive test scripts for Qt/nb issues
- #959: Matplotlib figures hide
- #2071: win32 installer issue on Windows XP
- #2610: ZMQInteractiveShell.colors being ignored
- #2505: Markdown Cell incorrectly highlighting after "<"
- #165: Installer fails to create Start Menu entries on Windows
- #2356: failing traceback in terminal ipython for first exception
- #2145: Have dashboad show when server disconect
- #2098: Do not crash on kernel shutdow if json file is missing
- #2813: Offline MathJax is broken on 0.14dev
- #2807: Test failure: IPython.parallel.tests.test_client.TestClient.test_purge_everything
- #2486: Readline's history search in ipython console does not clear properly after cancellation with Ctrl+C
- #2709: Cython -la doesn't work
- #2767: What is IPython.utils.upgradedir ?
- #2210: Placing matplotlib legend outside axis bounds causes inline display to clip it
- #2553: IPython Notebooks not robust against client failures
- #2536: ImageDraw in Ipython notebook not drawing lines
- #2264: Feature request: Versioning messaging protocol
- #2589: Creation of ~300+ MPI-spawned engines causes instability in ipcluster
- #2672: notebook: inline option without pylab
- #2673: Indefinite Articles & Traitlets
- #2705: Notebook crashes Safari with select and drag
- #2721: dreload kills ipython when it hits zmq

- #2806: ipython.parallel doesn't discover globals under Python 3.3
- #2794: _exit_code behaves differently in terminal vs ZMQ frontends
- #2793: IPython.parallel issue with pushing pandas TimeSeries
- #1085: In process kernel for Qt frontend
- #2760: IndexError: list index out of range with Python 3.2
- #2780: Save and load notebooks from github
- #2772: AttributeError: 'Client' object has no attribute 'kill'
- #2754: Fail to send class definitions from interactive session to engines namespaces
- #2764: TypeError while using 'cd'
- #2765: name '__file__' is not defined
- #2540: Wrap tooltip if line exceeds threshold?
- #2394: Startup error on ipython qtconsole (version 0.13 and 0.14-dev
- #2440: IPEP 4: Python 3 Compatibility
- #1814: __file__ is not defined when file end with .ipy
- #2759: R magic extension interferes with tab completion
- #2615: Small change needed to rmagic extension.
- #2748: collapse parts of a html notebook
- #1661: %paste still bugs about IndentationError and says to use %paste
- #2742: Octavemagic fails to deliver inline images in IPython (on Windows)
- #2739: wiki.ipython.org contaminated with prescription drug spam
- #2588: Link error while executing code from cython example notebook
- #2550: Rpush magic doesn't find local variables and doesn't support comma separated lists of variables
- #2675: Markdown/html blockquote need css.
- #2419: TerminalInteractiveShell.__init__() ignores value of ipython_dir argument
- #1523: Better LaTeX printing in the qtconsole with the sympy profile
- #2719: ipython fails with *pkg_resources.DistributionNotFound: ipython==0.13*
- #2715: url crashes nbviewer.ipython.org
- #2555: "import" module completion on MacOSX
- #2707: Problem installing the new version of IPython in Windows
- #2696: SymPy magic bug in IPython Notebook
- #2684: pretty print broken for types created with PyType_FromSpec

- #2533: rmagic breaks on Windows
- #2661: Qtconsole tooltip is too wide when the function has many arguments
- #2679: ipython3 qtconsole via Homebrew on Mac OS X 10.8 - pyqt/pyside import error
- #2646: pylab_not_importable
- #2587: cython magic pops 2 CLI windows upon execution on Windows
- #2660: Certain arguments (-h, –help, –version) never passed to scripts run with ipython
- #2665: Missing docs for rmagic and some other extensions
- #2611: Travis wants to drop 3.1 support
- #2658: Incorrect parsing of raw multiline strings
- #2655: Test fails if *from __future__ import print_function* in .pythonrc.py
- #2651: nonlocal with no existing variable produces too many errors
- #2645: python3 is a pain (minor unicode bug)
- #2637: %paste in Python 3 on Mac doesn't work
- #2624: Error on launching IPython on Win 7 and Python 2.7.3
- #2608: disk IO activity on cursor press
- #1275: Markdown parses LaTeX math symbols as its formatting syntax in notebook
- #2613: display(Math(...)) doesn't render tau correctly
- #925: Tab-completion in Qt console needn't use pager
- #2607: %load_ext sympy.interactive.ipythonprinting dammaging output
- #2593: Toolbar button to open qtconsole from notebook
- #2602: IPython html documentation for downloading
- #2598: ipython notebook –pylab=inline replaces built-in any()
- #2244: small issue: wrong printout
- #2590: add easier way to execute scripts in the current directory
- #2581: %hist does not work when InteractiveShell.cache_size = 0
- #2584: No file COPYING
- #2578: AttributeError: 'module' object has no attribute 'TestCase'
- #2576: One of my notebooks won't load any more – is there a maximum notebook size?
- #2560: Notebook output is invisible when printing strings with rrn line endings
- #2566: if pyside partially present ipython qtconsole fails to load even if pyqt4 present
- #1308: ipython qtconsole –ssh=server –existing ... hangs
- #1679: List command doesn't work in ipdb debugger the first time

- #2545: pypi win32 installer creates 64bit executibles
- #2080: Event loop issues with IPython 0.12 and PyQt4 (**QDialog.exec_** and more)
- #2541: Allow *python -m IPython*
- #2508: subplots_adjust() does not work correctly in ipython notebook
- #2289: Incorrect mathjax rendering of certain arrays of equations
- #2487: Selecting and indenting
- #2521: more fine-grained 'run' controls, such as 'run from here' and 'run until here'
- #2535: Funny bounding box when plot with text
- #2523: History not working
- #2514: Issue with zooming in qtconsole
- #2220: No sys.stdout.encoding in kernel based IPython
- #2512: ERROR: Internal Python error in the inspect module.
- #2496: Function passwd does not work in QtConsole
- #1453: make engines reconnect/die when controller was restarted
- #2481: ipython notebook – clicking in a code cell's output moves the screen to the top of the code cell
- #2488: Undesired plot outputs in Notebook inline mode
- #2482: ipython notebook – download may not get the latest notebook
- #2471: _subprocess module removed in Python 3.3
- #2374: Issues with man pages
- #2316: parallel.Client.__init__ should take cluster_id kwarg
- #2457: Can a R library wrapper be created with Rmagic?
- #1575: Fallback frontend for console when connecting pylab=inlnie -enabled kernel?
- #2097: Do not crash if history db is corrupted
- #2435: ipengines fail if clean_logs enabled
- #2429: Using warnings.warn() results in TypeError
- #2422: Multiprocessing in ipython notebook kernel crash
- #2426: ipython crashes with the following message. I do not what went wrong. Can you help me identify the problem?
- #2423: Docs typo?
- #2257: pip install -e fails
- #2418: rmagic can't run R's read.csv on data files with NA data
- #2417: HTML notebook: Backspace sometimes deletes multiple characters

- #2275: notebook: “Down_Arrow” on last line of cell should move to end of line
- #2414: 0.13.1 does not work with current EPD 7.3-2
- #2409: there is a redundant None
- #2410: Use /usr/bin/python3 instead of /usr/bin/python
- #2366: Notebook Dashboard –notebook-dir and fullpath
- #2406: Inability to get docstring in debugger
- #2398: Show line number for IndentationErrors
- #2314: HTML lists seem to interfere with the QtConsole display
- #1688: unicode exception when using %run with failing script
- #1884: IPython.embed changes color on error
- #2381: %time doesn’t work for multiline statements
- #1435: Add size keywords in Image class
- #2372: interactiveshell.py misses urllib and io_open imports
- #2371: iPython not working
- #2367: Tab expansion moves to next cell in notebook
- #2359: nbviever alters the order of print and display() output
- #2227: print name for IPython Notebooks has become uninformative
- #2361: client doesn’t use connection file’s ‘location’ in disambiguating ‘interface’
- #2357: failing traceback in terminal ipython for first exception
- #2343: Installing in a python 3.3b2 or python 3.3rc1 virtual environment.
- #2315: Failure in test: “Test we’re not loading modules on startup that we shouldn’t.”
- #2351: Multiple Notebook Apps: cookies not port specific, clash with each other
- #2350: running unittest from qtconsole prints output to terminal
- #2303: remote tracebacks broken since 952d0d6 (PR #2223)
- #2330: qtconsole does not hightlight tab-completion suggestion with custom stylesheet
- #2325: Parsing Tex formula fails in Notebook
- #2324: Parsing Tex formula fails
- #1474: Add argument to *run -n* for custom namespace
- #2318: C-m n/p don’t work in Markdown cells in the notebook
- #2309: time.time() in ipython notebook producing impossible results
- #2307: schedule tasks on newly arrived engines
- #2313: Allow Notebook HTML/JS to send messages to Python code

- #2304: ipengine throws KeyError: url
- #1878: shell access using ! will not fill class or function scope vars
- #2253: %paste does not retrieve clipboard contents under screen/tmux on OS X
- #1510: Add-on (or Monkey-patch) infrastructure for HTML notebook
- #2273: triple quote and %s at beginning of line with %paste
- #2243: Regression in .embed()
- #2266: SSH passwordless check with OpenSSH checks for the wrong thing
- #2217: Change NewNotebook handler to use 30x redirect
- #2276: config option for disabling history store
- #2239: can't use parallel.Reference in view.map
- #2272: Sympy piecewise messed up rendering
- #2252: %paste throws an exception with empty clipboard
- #2259: git-mpr is currently broken
- #2247: Variable expansion in shell commands should work in substrings
- #2026: Run 'fast' tests only
- #2241: read a list of notebooks on server and bring into browser only notebook
- #2237: please put python and text editor in the web only ipython
- #2053: Improvements to the IPython.display.Image object
- #1456: ERROR: Internal Python error in the inspect module.
- #2221: Avoid importing from IPython.parallel in core
- #2213: Can't trigger startup code in Engines
- #1464: Strange behavior for backspace with lines ending with more than 4 spaces in notebook
- #2187: NaN in object_info_reply JSON causes parse error
- #214: system command requiring administrative privileges
- #2195: Unknown option *no-edit* in git-mpr
- #2201: Add documentation build to tools/test_pr.py
- #2205: Command-line option for default Notebook output collapsing behavior
- #1927: toggle between inline and floating figures
- #2171: Can't start StarCluster after upgrading to IPython 0.13
- #2173: oct2py v >= 0.3.1 doesn't need h5py anymore
- #2099: storemagic needs to use self.shell
- #2166: DirectView map_sync() with Lambdas Using Generators

- #2091: Unable to use print_stats after %prun -r in notebook
- #2132: Add fail-over for pastebin
- #2156: Make it possible to install ipython without nasty gui dependencies
- #2154: Scrolled long output should be off in print view by default
- #2162: Tab completion does not work with IPython.embed_kernel()
- #2157: iPython 0.13 / github-master cannot create logfile from scratch
- #2151: missing newline when a magic is called from the qtconsole menu
- #2139: 00_notebook_tour Image example broken on master
- #2143: Add a %%cython_annotate magic
- #2135: Running IPython from terminal
- #2093: Makefile for building Sphinx documentation on Windows
- #2122: Bug in pretty printing
- #2120: Notebook "Make a Copy..." keeps opening duplicates in the same tab
- #1997: password cannot be used with url prefix
- #2129: help/doc displayed multiple times if requested in loop
- #2121: ipdb does not support input history in qtconsole
- #2114: %logstart doesn't log
- #2085: %ed magic fails in qtconsole
- #2119: iPython fails to run on MacOS Lion
- #2052: %pylab inline magic does not work on windows
- #2111: Ipython won't start on W7
- #2112: Strange internal traceback
- #2108: Backslash () at the end of the line behavior different from default Python
- #1425: Ampersands can't be typed sometimes in notebook cells
- #1513: Add expand/collapse support for long output elements like stdout and tracebacks
- #2087: error when starting ipython
- #2103: Ability to run notebook file from commandline
- #2082: Qt Console output spacing
- #2083: Test failures with Python 3.2 and PYTHONWARNINGS="d"
- #2094: about inline
- #2077: Starting IPython3 on the terminal
- #1760: easy_install ipython fails on py3.2-win32

- #2075: Local Mathjax install causes iptest3 error under python3
- #2057: setup fails for python3 with LANG=C
- #2070: shebang on Windows
- #2054: sys_info missing git hash in sdists
- #2059: duplicate and modified files in documentation
- #2056: except-shadows-builtin osm.py:687
- #2058: hyphen-used-as-minus-sign in manpages

2.4 0.13 Series

2.4.1 Release 0.13

IPython 0.13 contains several major new features, as well as a large amount of bug and regression fixes. The previous version (0.12) was released on December 19 2011, and in this development cycle we had:

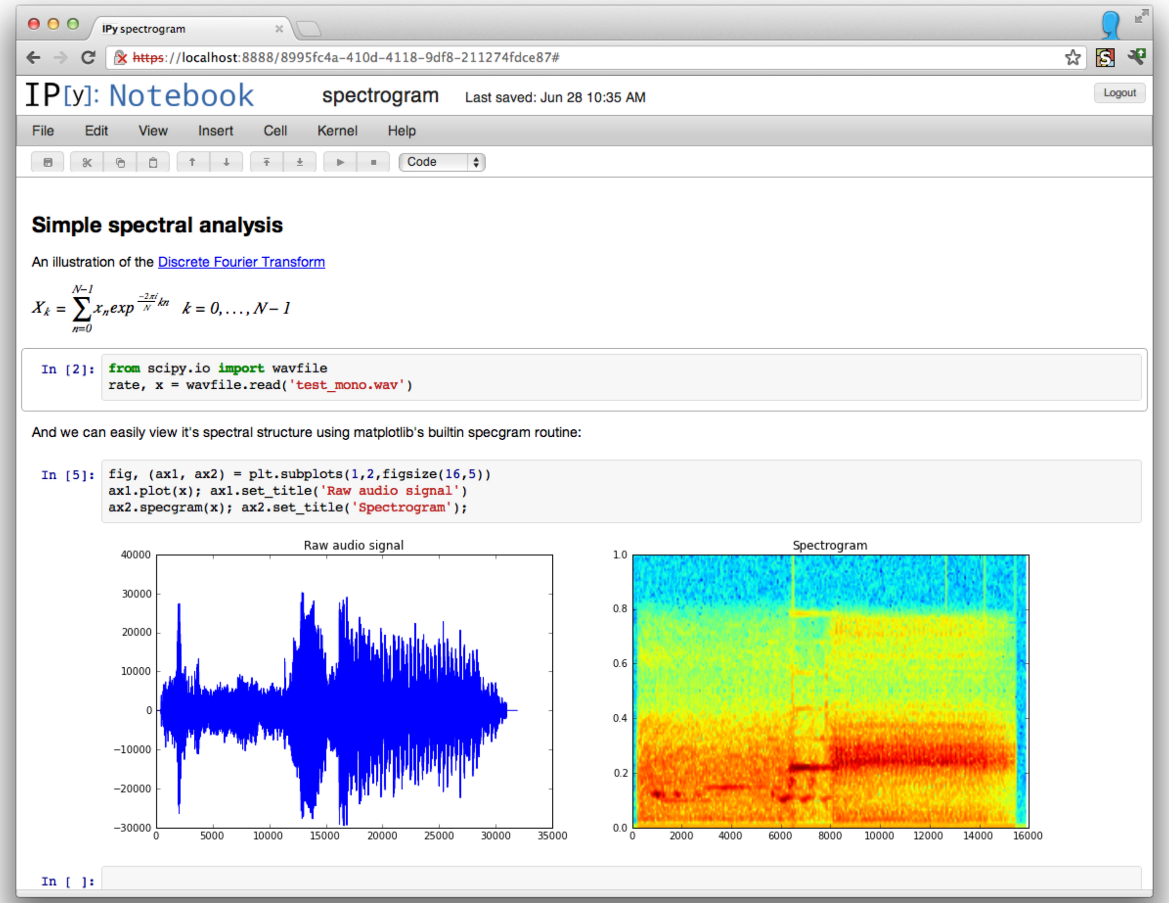
- ~6 months of work.
- 373 pull requests merged.
- 742 issues closed (non-pull requests).
- contributions from 62 authors.
- 1760 commits.
- a diff of 114226 lines.

The amount of work included in this release is so large, that we can only cover here the main highlights; please see our *detailed release statistics* for links to every issue and pull request closed on GitHub as well as a full list of individual contributors.

Major Notebook improvements: new user interface and more

The IPython Notebook, which has proven since its release to be wildly popular, has seen a massive amount of work in this release cycle, leading to a significantly improved user experience as well as many new features.

The first user-visible change is a reorganization of the user interface; the left panel has been removed and was replaced by a real menu system and a toolbar with icons. Both the toolbar and the header above the menu can be collapsed to leave an unobstructed working area:


IPy spectrogram
https://localhost:8888/8995fc4a-410d-4118-9df8-211274fdce87#
IP[y]: Notebook
spectrogram
Last saved: Jun 28 10:35 AM
Logout
File Edit View Insert Cell Kernel Help
Code

Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n exp^{\frac{-2\pi i}{N} kn} \quad k = 0, \ldots, N-1$$

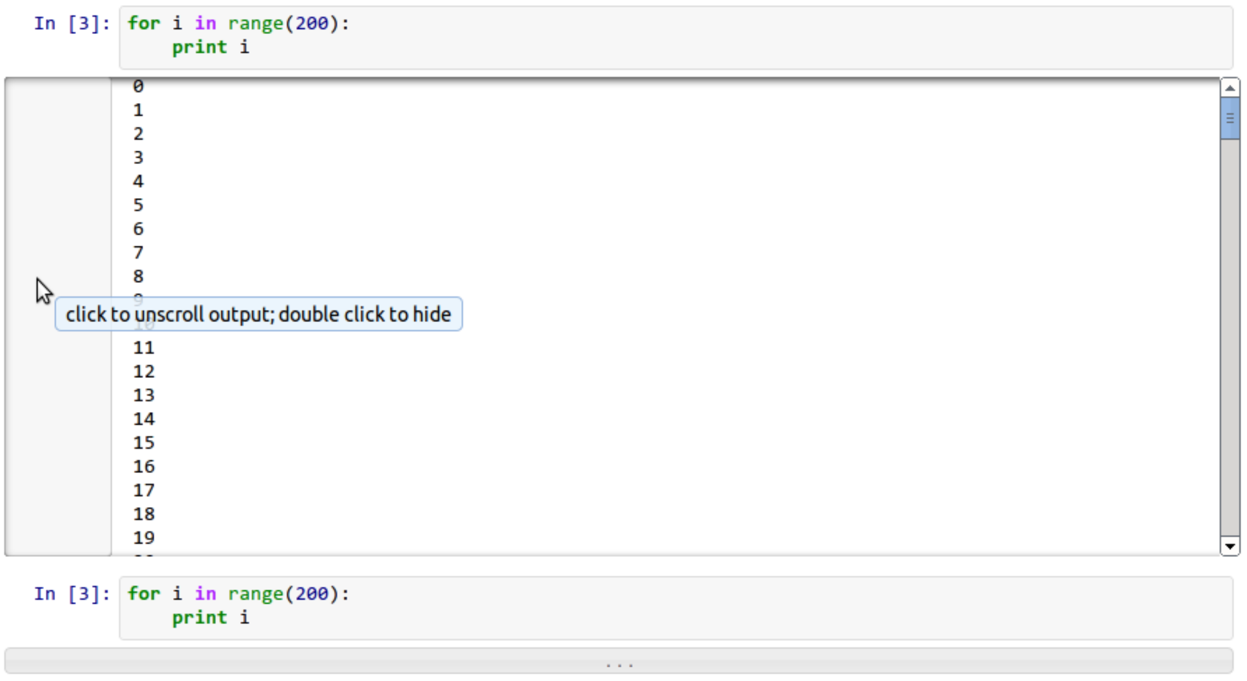
```
In [2]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view it's spectral structure using matplotlib's builtin specgram routine:

```
In [5]: fig, (ax1, ax2) = plt.subplots(1,2,figsize(16,5))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

```
In [ ]:
```

The notebook handles very long outputs much better than before (this was a serious usability issue when running processes that generated massive amounts of output). Now, in the presence of outputs longer than ~100 lines, the notebook will automatically collapse to a scrollable area and the entire left part of this area controls the display: one click in this area will expand the output region completely, and a double-click will hide it completely. This figure shows both the scrolled and hidden modes:

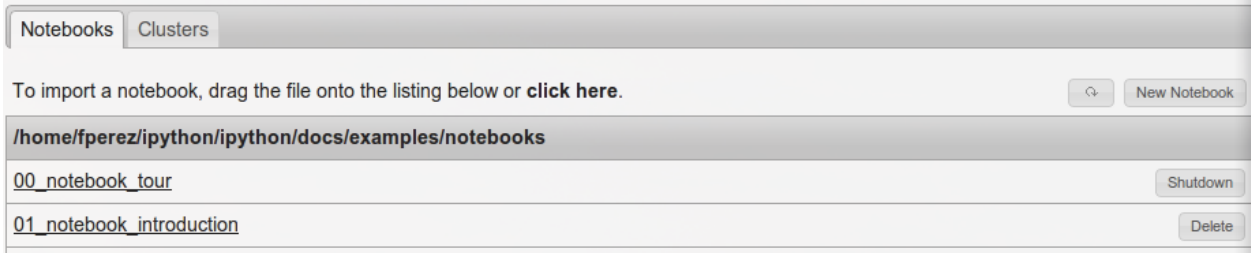


```
In [3]: for i in range(200):
            print i
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

```
In [3]: for i in range(200):
            print i
```

Note: The auto-folding of long outputs is disabled in Firefox due to bugs in its scrolling behavior. See PR #2047 for details.

Uploading notebooks to the dashboard is now easier: in addition to drag and drop (which can be finicky sometimes), you can now click on the upload text and use a regular file dialog box to select notebooks to upload. Furthermore, the notebook dashboard now auto-refreshes its contents and offers buttons to shut down any running kernels (PR #1739):



To import a notebook, drag the file onto the listing below or **click here**.

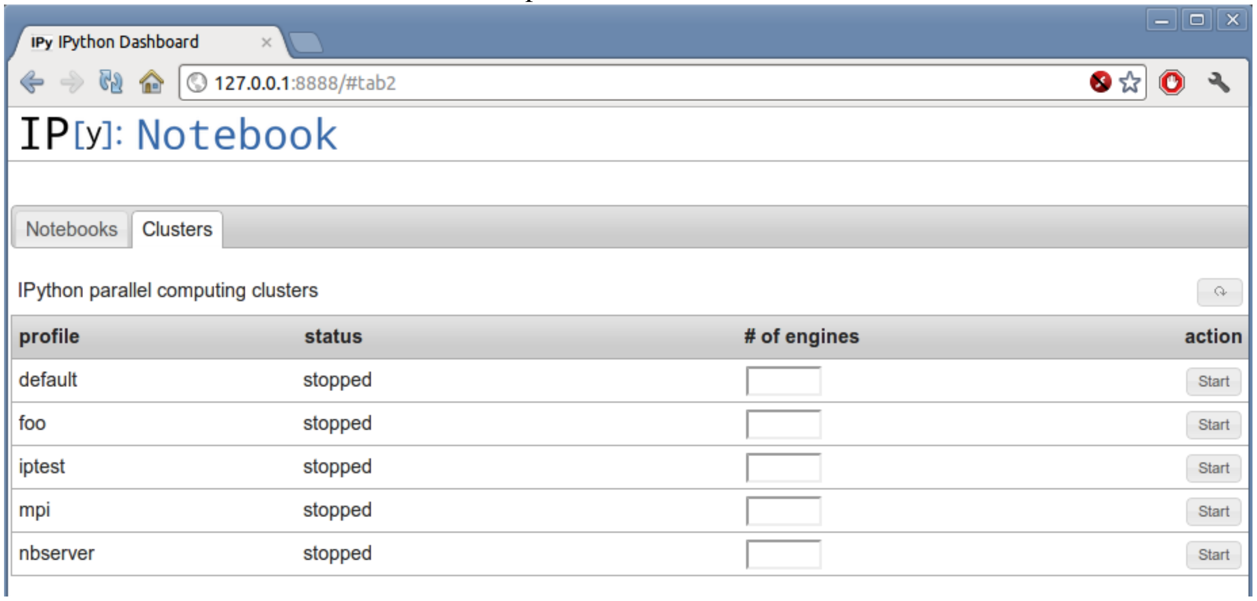
/home/fperez/ipython/ipython/docs/examples/notebooks

00_notebook_tour

01_notebook_introduction

Cluster management

The notebook dashboard can now also start and stop clusters, thanks to a new tab in the dashboard user inter-



IPython parallel computing clusters

profile	status	# of engines	action
default	stopped		Start
foo	stopped		Start
iptest	stopped		Start
mpi	stopped		Start
nbserver	stopped		Start

face:
This interface allows, for each profile you have configured, to start and stop a cluster (and optionally override the default number of engines corresponding to that configuration). While this hides all error reporting, once you have a configuration that you know works smoothly, it is a very convenient interface for controlling your parallel resources.

New notebook format

The notebooks saved now use version 3 of our format, which supports heading levels as well as the concept of 'raw' text cells that are not rendered as Markdown. These will be useful with converters we are developing, to pass raw markup (say LaTeX). That conversion code is still under heavy development and not quite ready for prime time, but we welcome help on this front so that we can merge it for full production use as soon as possible.

Note: v3 notebooks can *not* be read by older versions of IPython, but we provide a simple script that you can use in case you need to export a v3 notebook to share with a v2 user.

JavaScript refactoring

All the client-side JavaScript has been decoupled to ease reuse of parts of the machinery without having to build a full-blown notebook. This will make it much easier to communicate with an IPython kernel from existing web pages and to integrate single cells into other sites, without loading the full notebook document-like UI. PR #1711.

This refactoring also enables the possibility of writing dynamic javascript widgets that are returned from Python code and that present an interactive view to the user, with callbacks in Javascript executing calls to the Kernel. This will enable many interactive elements to be added by users in notebooks.

An example of this capability has been provided as a proof of concept in `examples/widgets` that lets you directly communicate with one or more parallel engines, acting as a mini-console for parallel debugging and introspection.

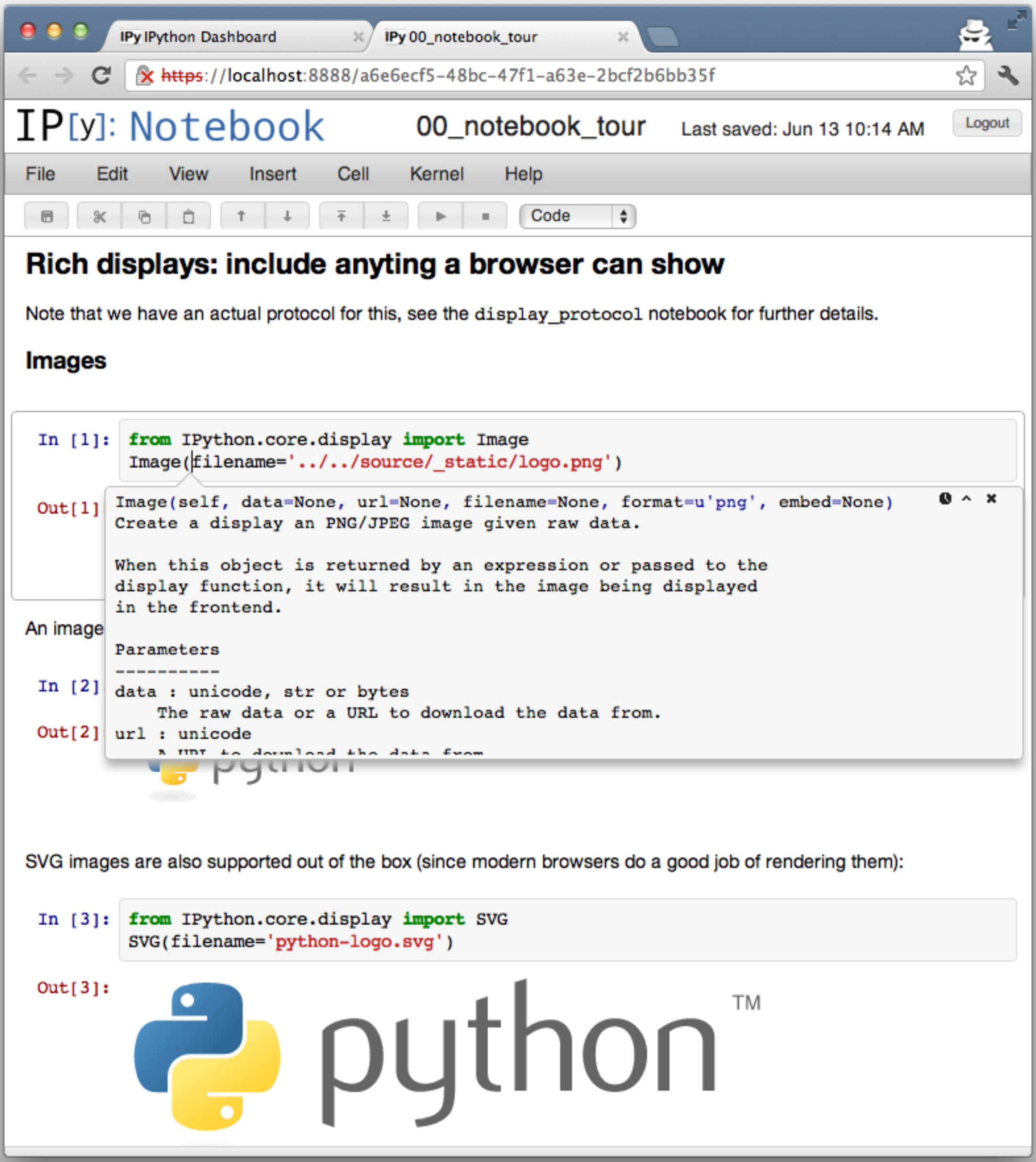
Improved tooltips

The object tooltips have gained some new functionality. By pressing tab several times, you can expand them to see more of a docstring, keep them visible as you fill in a function's parameters, or transfer the information to the pager at the bottom of the screen. For the details, look at the example notebook `01_notebook_introduction.ipynb`.

Other improvements to the Notebook

These are some other notable small improvements to the notebook, in addition to many bug fixes and minor changes to add polish and robustness throughout:

- The notebook pager (the area at the bottom) is now resizeable by dragging its divider handle, a feature that had been requested many times by just about anyone who had used the notebook system. PR #1705.
- It is now possible to open notebooks directly from the command line; for example: `ipython notebook path/` will automatically set `path/` as the notebook directory, and `ipython notebook path/foo.ipynb` will further start with the `foo.ipynb` notebook opened. PR #1686.
- If a notebook directory is specified with `--notebook-dir` (or with the corresponding configuration flag `NotebookManager.notebook_dir`), all kernels start in this directory.
- Fix codemirror clearing of cells with `Ctrl-Z`; PR #1965.
- Text (markdown) cells now line wrap correctly in the notebook, making them much easier to edit PR #1330.
- PNG and JPEG figures returned from plots can be interactively resized in the notebook, by dragging them from their lower left corner. PR #1832.
- Clear `In []` prompt numbers on "Clear All Output". For more version-control-friendly `.ipynb` files, we now strip all prompt numbers when doing a "Clear all output". This reduces the amount of noise in commit-to-commit diffs that would otherwise show the (highly variable) prompt number changes. PR #1621.
- The notebook server now requires *two* consecutive `Ctrl-C` within 5 seconds (or an interactive confirmation) to terminate operation. This makes it less likely that you will accidentally kill a long-running server by typing `Ctrl-C` in the wrong terminal. PR #1609.
- Using `Ctrl-S` (or `Cmd-S` on a Mac) actually saves the notebook rather than providing the fairly useless browser html save dialog. PR #1334.
- Allow accessing local files from the notebook (in urls), by serving any local file as the url `files/<relativepath>`. This makes it possible to, for example, embed local images in a notebook. PR #1211.



Rich displays: include anyting a browser can show

Note that we have an actual protocol for this, see the `display_protocol` notebook for further details.

Images

```
In [1]: from IPython.core.display import Image
        Image(filename='../../source/_static/logo.png')

Out[1]
```

```
Image(self, data=None, url=None, filename=None, format=u'png', embed=None)
Create a display an PNG/JPEG image given raw data.

When this object is returned by an expression or passed to the
display function, it will result in the image being displayed
in the frontend.

Parameters
----------
data : unicode, str or bytes
    The raw data or a URL to download the data from.
url : unicode
```

An image

```
In [2]
Out[2]
```

SVG images are also supported out of the box (since modern browsers do a good job of rendering them):

```
In [3]: from IPython.core.display import SVG
        SVG(filename='python-logo.svg')

Out[3]:
```

Fig. 2.1: The new notebook tooltips.

Cell magics

We have completely refactored the magic system, finally moving the magic objects to standalone, independent objects instead of being the mixin class we'd had since the beginning of IPython (PR #1732). Now, a separate base class is provided in `IPython.core.magic.Magics` that users can subclass to create their own magics. Decorators are also provided to create magics from simple functions without the need for object orientation. Please see the *Magic command system* docs for further details.

All builtin magics now exist in a few subclasses that group together related functionality, and the new `IPython.core.magics` package has been created to organize this into smaller files.

This cleanup was the last major piece of deep refactoring needed from the original 2001 codebase.

We have also introduced a new type of magic function, prefixed with %% instead of %, which operates at the whole-cell level. A cell magic receives two arguments: the line it is called on (like a line magic) and the body of the cell below it.

Cell magics are most natural in the notebook, but they also work in the terminal and qt console, with the usual approach of using a blank line to signal cell termination.

For example, to time the execution of several statements:

```
%%timeit x = 0   # setup
for i in range(100000):
    x += i**2
```

This is particularly useful to integrate code in another language, and cell magics already exist for shell scripts, Cython, R and Octave. Using `%%script /usr/bin/foo`, you can run a cell in any interpreter that accepts code via stdin.

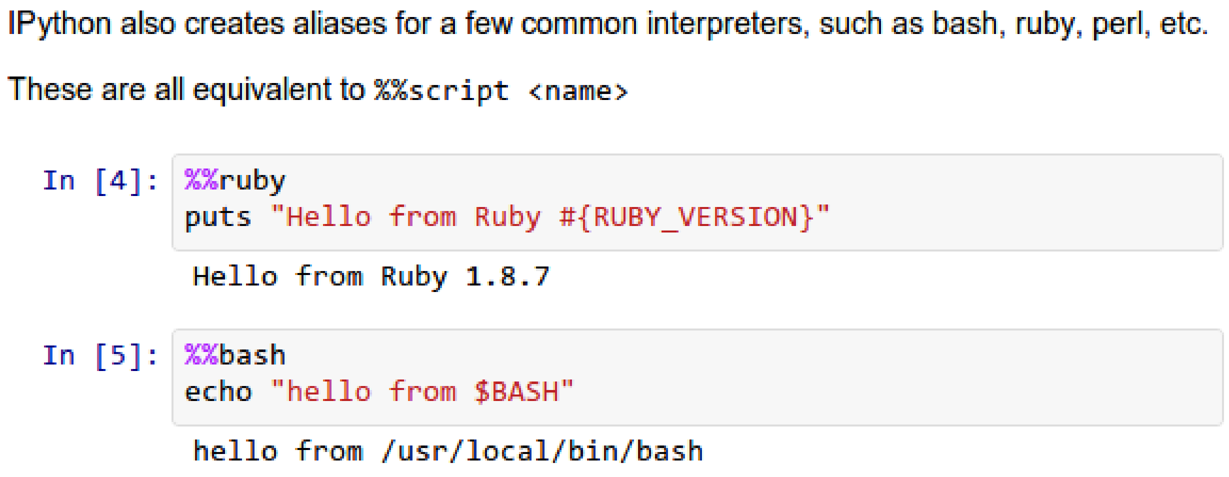
Another handy cell magic makes it easy to write short text files: `%%file ~/save/to/here.txt`.

The following cell magics are now included by default; all those that use special interpreters (Perl, Ruby, bash, etc.) assume you have the requisite interpreter installed:

- `%%!`: run cell body with the underlying OS shell; this is similar to prefixing every line in the cell with `!`.
- `%%bash`: run cell body under bash.
- `%%capture`: capture the output of the code in the cell (and stderr as well). Useful to run codes that produce too much output that you don't even want scrolled.
- `%%file`: save cell body as a file.
- `%%perl`: run cell body using Perl.
- `%%prun`: run cell body with profiler (cell extension of `%prun`).
- `%%python3`: run cell body using Python 3.
- `%%ruby`: run cell body using Ruby.
- `%%script`: run cell body with the script specified in the first line.
- `%%sh`: run cell body using sh.
- `%%sx`: run cell with system shell and capture process output (cell extension of `%sx`).

- `%%system`: run cell with system shell (`%%!` is an alias to this).
- `%%timeit`: time the execution of the cell (extension of `%timeit`).

This is what some of the script-related magics look like in action:



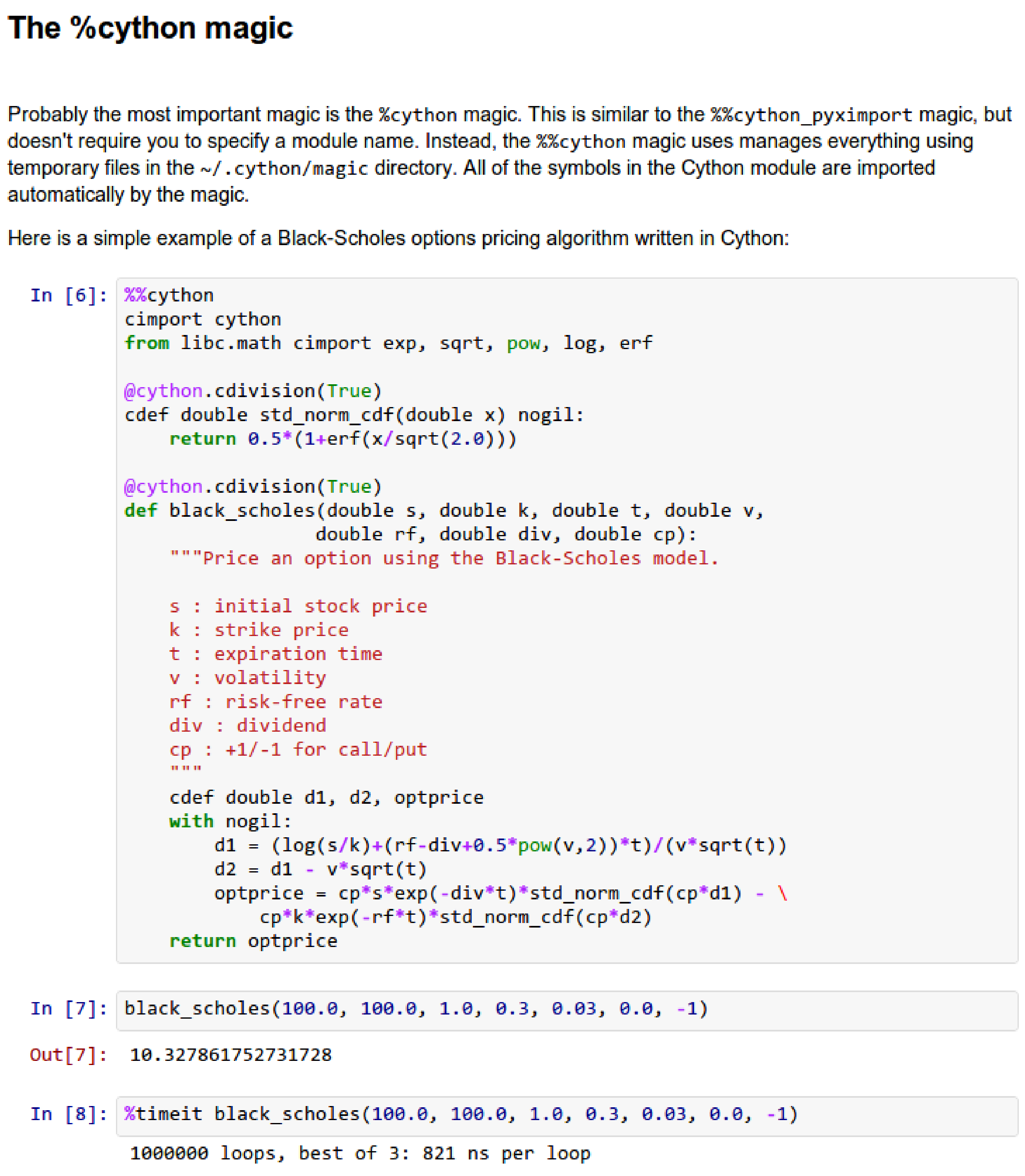
IPython also creates aliases for a few common interpreters, such as bash, ruby, perl, etc.

These are all equivalent to `%%script <name>`

```
In [4]: %%ruby
        puts "Hello from Ruby #{RUBY_VERSION}"

        Hello from Ruby 1.8.7

In [5]: %%bash
        echo "hello from $BASH"

        hello from /usr/local/bin/bash
```

In addition, we have also a number of *extensions* that provide specialized magics. These typically require additional software to run and must be manually loaded via `%load_ext <extension name>`, but are extremely useful. The following extensions are provided:

Cython magics (extension *cythonmagic*) This extension provides magics to automatically build and compile Python extension modules using the Cython language. You must install Cython separately, as well as a C compiler, for this to work. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:



The %cython magic

Probably the most important magic is the `%cython` magic. This is similar to the `%%cython_pyximport` magic, but doesn't require you to specify a module name. Instead, the `%%cython` magic uses manages everything using temporary files in the `~/.cython/magic` directory. All of the symbols in the Cython module are imported automatically by the magic.

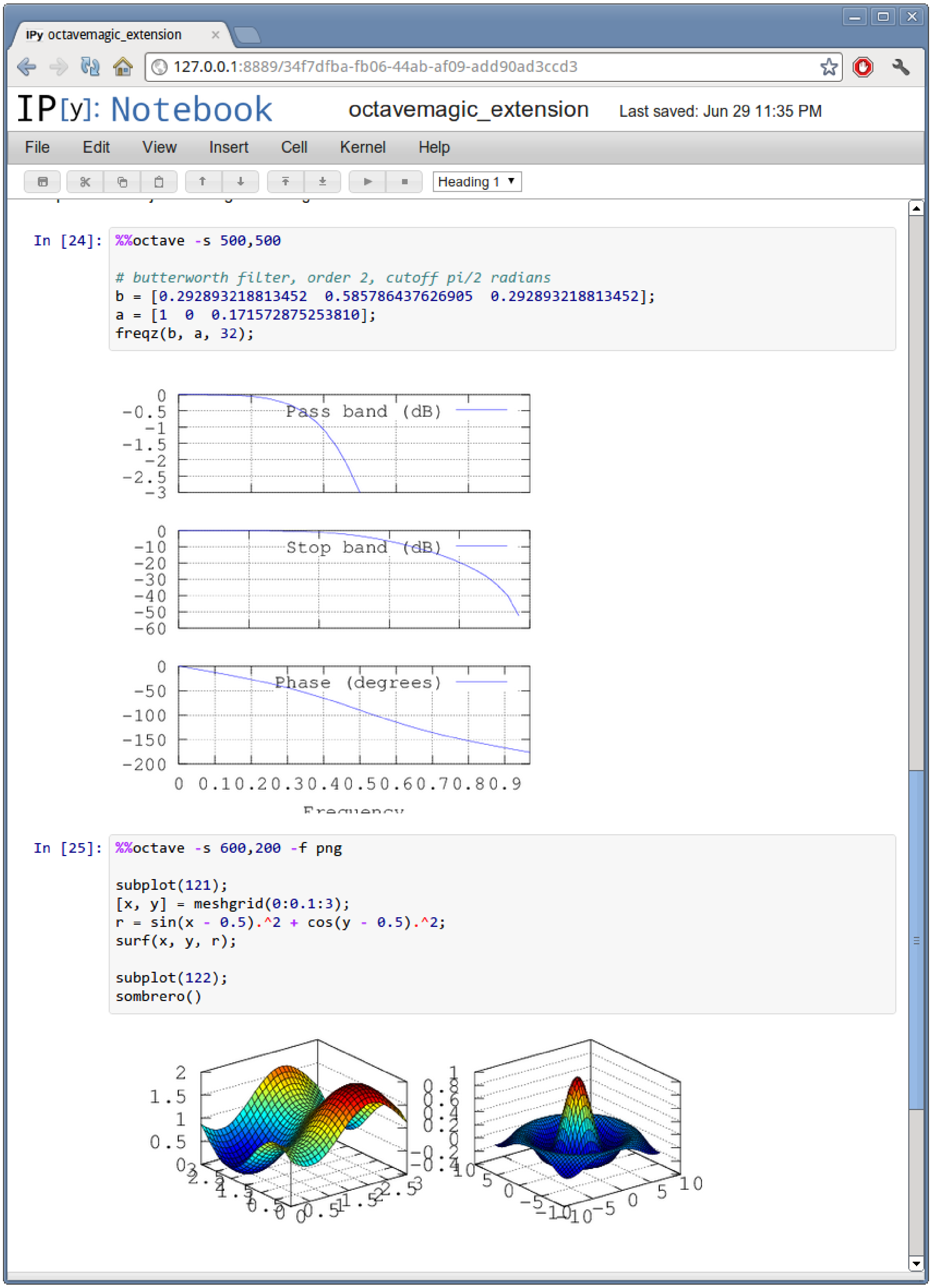
Here is a simple example of a Black-Scholes options pricing algorithm written in Cython:

```
In [6]: %%cython
        cimport cython
        from libc.math cimport exp, sqrt, pow, log, erf

        @cython.cdivision(True)
        cdef double std_norm_cdf(double x) nogil:
            return 0.5*(1+erf(x/sqrt(2.0)))

        @cython.cdivision(True)
        def black_scholes(double s, double k, double t, double v,
                         double rf, double div, double cp):
            """Price an option using the Black-Scholes model.

            s : initial stock price
            k : strike price
            t : expiration time
            v : volatility
            rf : risk-free rate
            div : dividend
            cp : +1/-1 for call/put
            """
            cdef double d1, d2, optprice
            with nogil:
                d1 = (log(s/k)+(rf-div+0.5*pow(v,2))*t)/(v*sqrt(t))
                d2 = d1 - v*sqrt(t)
                optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1) - \
                    cp*k*exp(-rf*t)*std_norm_cdf(cp*d2)
            return optprice
```

```
In [7]: black_scholes(100.0, 100.0, 1.0, 0.3, 0.03, 0.0, -1)

Out[7]:  10.327861752731728
```

```
In [8]: %timeit black_scholes(100.0, 100.0, 1.0, 0.3, 0.03, 0.0, -1)

         1000000 loops, best of 3: 821 ns per loop
```

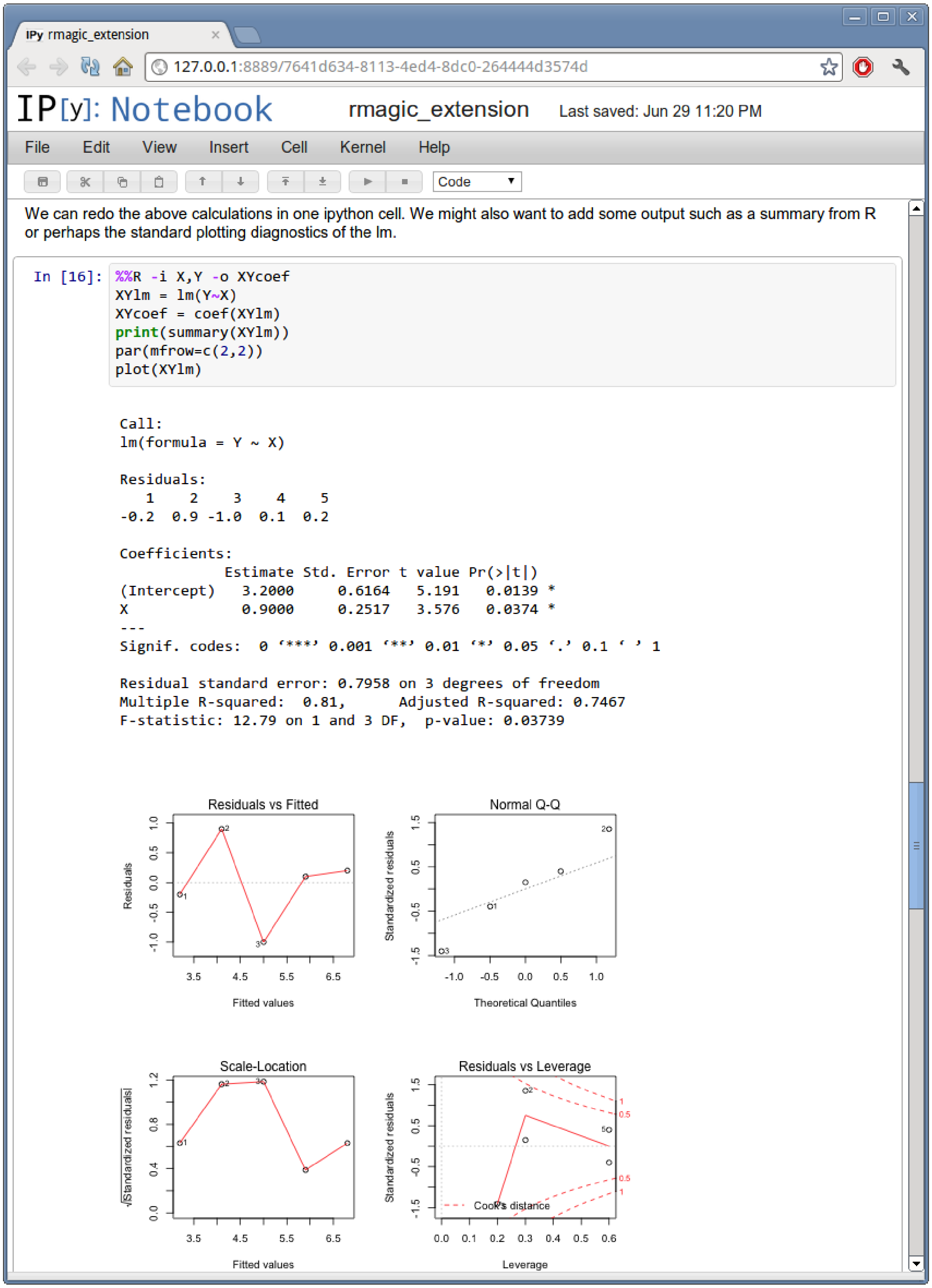
Octave magics (extension *octavemagic*) This extension provides several magics that support calling code written in the Octave language for numerical computing. You can execute single-lines or whole blocks of Octave code, capture both output and figures inline (just like matplotlib plots), and have variables automatically converted between the two languages. To use this extension, you must have Octave installed as well as the oct2py package. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:



```
In [24]: %%octave -s 500,500

         # butterworth filter, order 2, cutoff pi/2 radians
         b = [0.292893218813452  0.585786437626905  0.292893218813452];
         a = [1  0  0.171572875253810];
         freqz(b, a, 32);
```

```
In [25]: %%octave -s 600,200 -f png

         subplot(121);
         [x, y] = meshgrid(0:0.1:3);
         r = sin(x - 0.5).^2 + cos(y - 0.5).^2;
         surf(x, y, r);

         subplot(122);
         sombrero()
```

R magics (extension *rmagic*) This extension provides several magics that support calling code written in the R language for statistical data analysis. You can execute single-lines or whole blocks of R code, capture both output and figures inline (just like matplotlib plots), and have variables automatically converted between the two languages. To use this extension, you must have R installed as well as the rpy2 package that bridges Python and R. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:



We can redo the above calculations in one ipython cell. We might also want to add some output such as a summary from R or perhaps the standard plotting diagnostics of the lm.

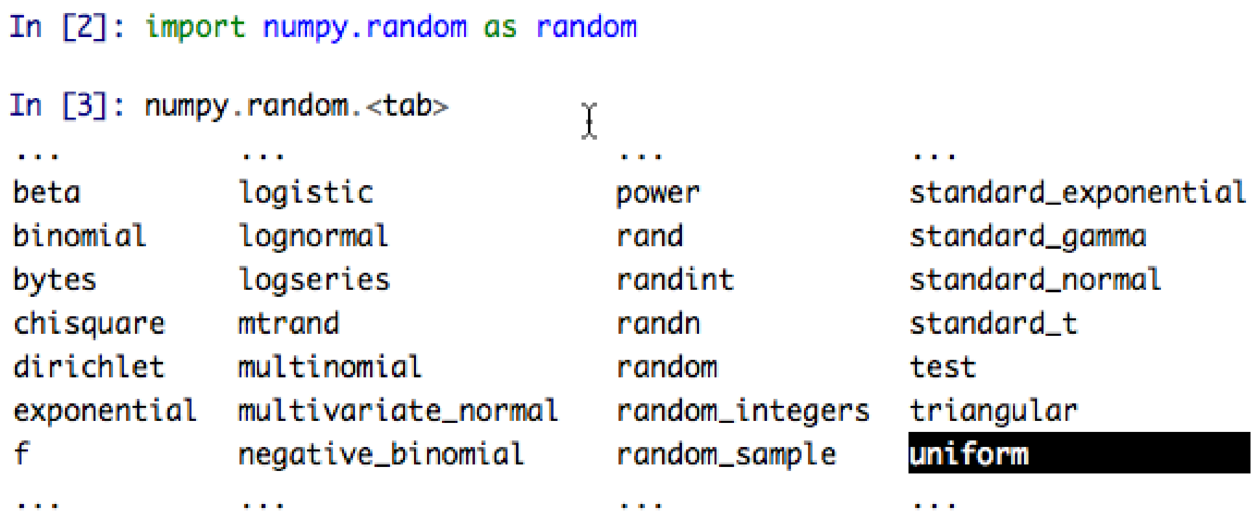
```
In [16]: %%R -i X,Y -o XYcoef
         XYlm = lm(Y~X)
         XYcoef = coef(XYlm)
         print(summary(XYlm))
         par(mfrow=c(2,2))
         plot(XYlm)
```

```
Call:
lm(formula = Y ~ X)

Residuals:
   1    2    3    4    5 
-0.2  0.9 -1.0  0.1  0.2 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)  
(Intercept)   3.2000     0.6164   5.191   0.0139 *
X             0.9000     0.2517   3.576   0.0374 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.7958 on 3 degrees of freedom
Multiple R-squared:  0.81,      Adjusted R-squared: 0.7467 
F-statistic: 12.79 on 1 and 3 DF,  p-value: 0.03739
```

Tab completer improvements

Useful tab-completion based on live inspection of objects is one of the most popular features of IPython. To make this process even more user-friendly, the completers of both the Qt console and the Notebook have been reworked.

The Qt console comes with a new ncurses-like tab completer, activated by default, which lets you cycle through the available completions by pressing tab, or select a completion with the arrow keys (PR #1851).



```
In [2]: import numpy.random as random

In [3]: numpy.random.<tab>
...            ...                    ...               ...
beta           logistic               power             standard_exponential
binomial       lognormal              rand              standard_gamma
bytes          logseries              randint           standard_normal
chisquare      mtrand                 randn             standard_t
dirichlet      multinomial            random            test
exponential    multivariate_normal    random_integers   triangular
f              negative_binomial      random_sample     uniform
...            ...                    ...               ...
```

Fig. 2.2: The new improved Qt console's ncurses-like completer allows to easily navigate thought long list of completions.

In the notebook, completions are now sourced both from object introspection and analysis of surrounding code, so limited completions can be offered for variables defined in the current cell, or while the kernel is busy (PR #1711).

We have implemented a new configurable flag to control tab completion on modules that provide the `__all__` attribute:

```
IPCompleter.limit_to__all__= Boolean
```

This instructs the completer to honor `__all__` for the completion. Specifically, when completing on `object.<tab>`, if True: only those names in `obj.__all__` will be included. When False [default]: the `__all__` attribute is ignored. PR #1529.

Improvements to the Qt console

The Qt console continues to receive improvements and refinements, despite the fact that it is by now a fairly mature and robust component. Lots of small polish has gone into it, here are a few highlights:

- A number of changes were made to the underlying code for easier integration into other projects such as Spyder (PR #2007, PR #2024).

- Improved menus with a new Magic menu that is organized by magic groups (this was made possible by the reorganization of the magic system internals). PR #1782.
- Allow for restarting kernels without clearing the qtconsole, while leaving a visible indication that the kernel has restarted. PR #1681.
- Allow the native display of jpeg images in the qtconsole. PR #1643.

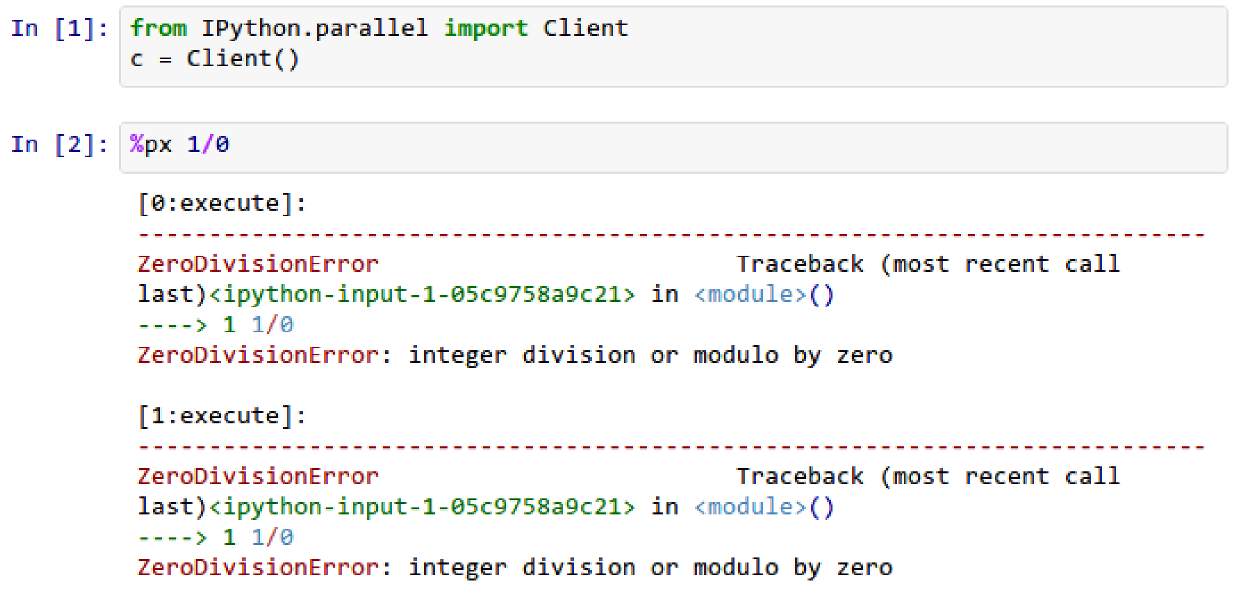
Parallel

The parallel tools have been improved and fine-tuned on multiple fronts. Now, the creation of an `IPython.parallel.Client` object automatically activates a line and cell magic function `px` that sends its code to all the engines. Further magics can be easily created with the `Client.activate()` method, to conveniently execute code on any subset of engines. PR #1893.

The `%%px` cell magic can also be given an optional targets argument, as well as a `--out` argument for storing its output.

A new magic has also been added, `%pxconfig`, that lets you configure various defaults of the parallel magics. As usual, type `%pxconfig?` for details.

The exception reporting in parallel contexts has been improved to be easier to read. Now, IPython directly reports the remote exceptions without showing any of the internal execution parts:



```
In [1]: from IPython.parallel import Client
        c = Client()

In [2]: %px 1/0

        [0:execute]:
        ---------------------------------------------------------------------------
        ZeroDivisionError                         Traceback (most recent call
        last)<ipython-input-1-05c9758a9c21> in <module>()
        ----> 1 1/0
        ZeroDivisionError: integer division or modulo by zero

        [1:execute]:
        ---------------------------------------------------------------------------
        ZeroDivisionError                         Traceback (most recent call
        last)<ipython-input-1-05c9758a9c21> in <module>()
        ----> 1 1/0
        ZeroDivisionError: integer division or modulo by zero
```

The parallel tools now default to using `NoDB` as the storage backend for intermediate results. This means that the default usage case will have a significantly reduced memory footprint, though certain advanced features are not available with this backend. For more details, see *IPython's Task Database*.

The parallel magics now display all output, so you can do parallel plotting or other actions with complex display. The `px` magic has now both line and cell modes, and in cell mode finer control has been added about how to collate output from multiple engines. PR #1768.

There have also been incremental improvements to the SSH launchers:

- add to_send/fetch steps for moving connection files around.

- add SSHProxyEngineSetLauncher, for invoking to *ipcluster engines* on a remote host. This can be used to start a set of engines via PBS/SGE/MPI *remotely*.

This makes the SSHLauncher usable on machines without shared filesystems.

A number of 'sugar' methods/properties were added to AsyncResult that are quite useful (PR #1548) for everday work:

- `ar.wall_time` = received - submitted
- `ar.serial_time` = sum of serial computation time
- `ar.elapsed` = time since submission (wall_time if done)
- `ar.progress` = (int) number of sub-tasks that have completed
- `len(ar)` = # of tasks
- `ar.wait_interactive()`: prints progress

Added `Client.spin_thread()` / `stop_spin_thread()` for running spin in a background thread, to keep zmq queue clear. This can be used to ensure that timing information is as accurate as possible (at the cost of having a background thread active).

Set TaskScheduler.hwm default to 1 instead of 0. 1 has more predictable/intuitive behavior, if often slower, and thus a more logical default. Users whose workloads require maximum throughput and are largely homogeneous in time per task can make the optimization themselves, but now the behavior will be less surprising to new users. PR #1294.

Kernel/Engine unification

This is mostly work 'under the hood', but it is actually a *major* achievement for the project that has deep implications in the long term: at last, we have unified the main object that executes as the user's interactive shell (which we refer to as the *IPython kernel*) with the objects that run in all the worker nodes of the parallel computing facilities (the *IPython engines*). Ever since the first implementation of IPython's parallel code back in 2006, we had wanted to have these two roles be played by the same machinery, but a number of technical reasons had prevented that from being true.

In this release we have now merged them, and this has a number of important consequences:

- It is now possible to connect any of our clients (qtconsole or terminal console) to any individual parallel engine, with the *exact* behavior of working at a 'regular' IPython console/qtconsole. This makes debugging, plotting, etc. in parallel scenarios vastly easier.
- Parallel engines can always execute arbitrary 'IPython code', that is, code that has magics, shell extensions, etc. In combination with the `%%px` magics, it is thus extremely natural for example to send to all engines a block of Cython or R code to be executed via the new Cython and R magics. For example, this snippet would send the R block to all active engines in a cluster:

```
%%px
%%R
... R code goes here
```

- It is possible to embed not only an interactive shell with the `IPython.embed()` call as always, but now you can also embed a *kernel* with `IPython.embed_kernel()`. Embedding an IPython kernel in an application is useful when you want to use `IPython.embed()` but don't have a terminal attached on stdin and stdout.
- The new `IPython.parallel.bind_kernel()` allows you to promote Engines to listening Kernels, and connect QtConsoles to an Engine and debug it directly.

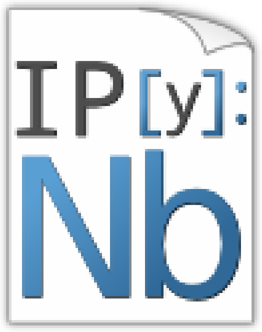
In addition, having a single core object through our entire architecture also makes the project conceptually cleaner, easier to maintain and more robust. This took a lot of work to get in place, but we are thrilled to have this major piece of architecture finally where we'd always wanted it to be.

Official Public API

We have begun organizing our API for easier public use, with an eye towards an official IPython 1.0 release which will firmly maintain this API compatible for its entire lifecycle. There is now an `IPython.display` module that aggregates all display routines, and the `IPython.config` namespace has all public configuration tools. We will continue improving our public API layout so that users only need to import names one level deeper than the main `IPython` package to access all public namespaces.

IPython notebook file icons

The directory `docs/resources` in the source distribution contains SVG and PNG versions of our file icons, as well as an `Info.plist.example` file with instructions to install them on Mac OSX. This is a first draft of our icons, and we encourage contributions from users with graphic talent to improve them in the future:



New top-level *locate* command

Add *locate* entry points; these would be useful for quickly locating IPython directories and profiles from other (non-Python) applications. PR #1762.

Examples:

```
$> ipython locate
/Users/me/.ipython

$> ipython locate profile foo
```

```
/Users/me/.ipython/profile_foo

$> ipython locate profile
/Users/me/.ipython/profile_default

$> ipython locate profile dne
[ProfileLocate] Profile u'dne' not found.
```

Other new features and improvements

- **%install_ext**: A new magic function to install an IPython extension from a URL. E.g. `%install_ext https://bitbucket.org/birkenfeld/ipython-physics/raw/default/phys`
- The `%loadpy` magic is no longer restricted to Python files, and has been renamed `%load`. The old name remains as an alias.
- New command line arguments will help external programs find IPython folders: `ipython locate` finds the user's IPython directory, and `ipython locate profile foo` finds the folder for the 'foo' profile (if it exists).
- The `IPYTHON_DIR` environment variable, introduced in the Great Reorganization of 0.11 and existing only in versions 0.11-0.13, has been deprecated. As described in PR #1167, the complexity and confusion of migrating to this variable is not worth the aesthetic improvement. Please use the historical `IPYTHONDIR` environment variable instead.
- The default value of *interactivity* passed from `run_cell()` to `run_ast_nodes()` is now configurable.
- New `%alias_magic` function to conveniently create aliases of existing magics, if you prefer to have shorter names for personal use.
- We ship unminified versions of the JavaScript libraries we use, to better comply with Debian's packaging policies.
- Simplify the information presented by `obj?/obj??` to eliminate a few redundant fields when possible. PR #2038.
- Improved continuous integration for IPython. We now have automated test runs on Shining Panda and Travis-CI, as well as Tox support.
- The vim-ipython functionality (externally developed) has been updated to the latest version.
- The `%save` magic now has a `-f` flag to force overwriting, which makes it much more usable in the notebook where it is not possible to reply to interactive questions from the kernel. PR #1937.
- Use dvipng to format sympy.Matrix, enabling display of matrices in the Qt console with the sympy printing extension. PR #1861.
- Our messaging protocol now has a reasonable test suite, helping ensure that we don't accidentally deviate from the spec and possibly break third-party applications that may have been using it. We encourage users to contribute more stringent tests to this part of the test suite. PR #1627.
- Use LaTeX to display, on output, various built-in types with the SymPy printing extension. PR #1399.

- Add Gtk3 event loop integration and example. PR #1588.
- `clear_output` improvements, which allow things like progress bars and other simple animations to work well in the notebook (PR #1563):
  - *clear_output()* clears the line, even in terminal IPython, the QtConsole and plain Python as well, by printing *r* to streams.
  - *clear_output()* avoids the flicker in the notebook by adding a delay, and firing immediately upon the next actual display message.
  - *display_javascript* hides its *output_area* element, so using display to run a bunch of javascript doesn't result in ever-growing vertical space.
- Add simple support for running inside a virtualenv. While this doesn't supplant proper installation (as users should do), it helps ad-hoc calling of IPython from inside a virtualenv. PR #1388.

Major Bugs fixed

In this cycle, we have *closed over 740 issues*, but a few major ones merit special mention:

- The `%pastebin` magic has been updated to point to gist.github.com, since unfortunately http://paste.pocoo.org has closed down. We also added a -d flag for the user to provide a gist description string. PR #1670.
- Fix `%paste` that would reject certain valid inputs. PR #1258.
- Fix sending and receiving of Numpy structured arrays (those with composite dtypes, often used as recarrays). PR #2034.
- Reconnect when the websocket connection closes unexpectedly. PR #1577.
- Fix truncated representation of objects in the debugger by showing at least 80 characters' worth of information. PR #1793.
- Fix logger to be Unicode-aware: logging could crash ipython if there was unicode in the input. PR #1792.
- Fix images missing from XML/SVG export in the Qt console. PR #1449.
- Fix deepreload on Python 3. PR #1625, as well as having a much cleaner and more robust implementation of deepreload in general. PR #1457.

Backwards incompatible changes

- The exception `IPython.core.error.TryNext` previously accepted arguments and keyword arguments to be passed to the next implementation of the hook. This feature was removed as it made error message propagation difficult and violated the principle of loose coupling.

2.5 Issues closed in the 0.13 development cycle

2.5.1 Issues closed in 0.13

GitHub stats since IPython 0.12 (2011/12/19 - 2012/06/30)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 62 authors contributed 1760 commits.

- Aaron Culich
- Aaron Meurer
- Alex Kramer
- Andrew Giessel
- Andrew Straw
- André Matos
- Aron Ahmadia
- Ben Edwards
- Benjamin Ragan-Kelley
- Bradley M. Froehle
- Brandon Parsons
- Brian E. Granger
- Carlos Cordoba
- David Hirschfeld
- David Zderic
- Ernie French
- Fernando Perez
- Ian Murray
- Jason Grout
- Jens H Nielsen
- Jez Ng
- Jonathan March
- Jonathan Taylor
- Julian Taylor
- Jörgen Stenarson
- Kent Inverarity

- Marc Abramowitz
- Mark Wiebe
- Matthew Brett
- Matthias BUSSONNIER
- Michael Droettboom
- Mike Hansen
- Nathan Rice
- Pankaj Pandey
- Paul
- Paul Ivanov
- Piotr Zolnierczuk
- Piti Ongmongkolkul
- Puneeth Chaganti
- Robert Kern
- Ross Jones
- Roy Hyunjin Han
- Scott Tsai
- Skipper Seabold
- Stefan van der Walt
- Steven Johnson
- Takafumi Arakaki
- Ted Wright
- Thomas Hisch
- Thomas Kluyver
- Thomas Spura
- Thomi Richards
- Tim Couper
- Timo Paulssen
- Toby Gilham
- Tony S Yu
- 23. Trevor King
- Walter Doerwald

- anatoly techtonik
- fawce
- mcelrath
- wilsaj

We closed a total of 1115 issues, 373 pull requests and 742 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (373):

- PR #1943: add screenshot and link into releasenotes
- PR #1954: update some example notebooks
- PR #2048: move _encode_binary to jsonutil.encode_images
- PR #2050: only add quotes around xunit-file on Windows
- PR #2047: disable auto-scroll on mozilla
- PR #2015: Fixes for %paste with special transformations
- PR #2046: Iptest unicode
- PR #1939: Namespaces
- PR #2042: increase auto-scroll threshold to 100 lines
- PR #2043: move RemoteError import to top-level
- PR #2036: %alias_magic
- PR #1968: Proposal of icons for .ipynb files
- PR #2037: remove *ipython-qtconsole* gui-script
- PR #2038: add extra clear warning to shell doc
- PR #2029: Ship unminified js
- PR #2007: Add custom_control and custom_page_control variables to override the Qt widgets used by qtconsole
- PR #2034: fix&test push/pull recarrays
- PR #2028: Reduce unhelpful information shown by pinfo
- PR #2030: check wxPython version in inputhook
- PR #2024: Make interactive_usage a bit more rst friendly
- PR #2031: disable ^C^C confirmation on Windows
- PR #2027: match stdin encoding in frontend readline test
- PR #2025: Fix parallel test on WinXP - wait for resource cleanup.
- PR #2016: BUG: test runner fails in Windows if filenames contain spaces.
- PR #2020: Fix home path expansion test in Windows.

- PR #2021: Fix Windows pathname issue in 'odd encoding' test.
- PR #2022: don't check writability in test for get_home_dir when HOME is undefined
- PR #1996: frontend test tweaks
- PR #2014: relax profile regex in notebook
- PR #2012: Mono cursor offset
- PR #2004: Clarify generic message spec vs. Python message API in docs
- PR #2010: notebook: Print a warning (but do not abort) if no webbrowser can be found.
- PR #2002: Refactor %magic into a lsmagic_docs API function.
- PR #1999: *%magic* help: display line and cell magics in alphabetical order.
- PR #1981: Clean BG processes created by %%script on kernel exit
- PR #1994: Fix RST misformatting.
- PR #1951: minor notebook startup/notebook-dir adjustments
- PR #1974: Allow path completion on notebook.
- PR #1964: allow multiple instances of a Magic
- PR #1991: fix _ofind attr in %page
- PR #1988: check for active frontend in update_restart_checkbox
- PR #1979: Add support for tox (http://tox.testrun.org/) and Travis CI (http://travis-ci.org/)
- PR #1970: dblclick to restore size of images
- PR #1978: Notebook names truncating at the first period
- PR #1825: second attempt at scrolled long output
- PR #1934: Cell/Worksheet metadata
- PR #1746: Confirm restart (configuration option, and checkbox UI)
- PR #1944: [qtconsole] take %,%% prefix into account for completion
- PR #1973: fix another FreeBSD $HOME symlink issue
- PR #1967: Fix psums example description in docs
- PR #1965: fix for #1678, undo no longer clears cells
- PR #1952: avoid duplicate "Websockets closed" dialog on ws close
- PR #1962: Support unicode prompts
- PR #1955: update to latest version of vim-ipython
- PR #1945: Add –proc option to %%script
- PR #1956: move import RemoteError after get_exc_info
- PR #1950: Fix for copy action (Ctrl+C) when there is no pager defined in qtconsole

- PR #1948: Fix help string for InteractiveShell.ast_node_interactivity
- PR #1942: swallow stderr of which in utils.process.find_cmd
- PR #1940: fix completer css on some Chrome versions
- PR #1938: remove remaining references to deprecated XREP/XREQ names
- PR #1925: Fix styling of superscripts and subscripts. Closes #1924.
- PR #1936: increase duration of save messages
- PR #1937: add %save -f
- PR #1935: add version checking to pyreadline import test
- PR #1849: Octave magics
- PR #1759: github, merge PR(s) just by number(s)
- PR #1931: Win py3fixes
- PR #1933: oinspect.find_file: Additional safety if file cannot be found.
- PR #1932: Fix adding functions to CommandChainDispatcher with equal priority on Py 3
- PR #1928: Select NoDB by default
- PR #1923: Add IPython syntax support to the %timeit magic, in line and cell mode
- PR #1926: Make completer recognize escaped quotes in strings.
- PR #1893: Update Parallel Magics and Exception Display
- PR #1921: magic_arguments: dedent but otherwise preserve indentation.
- PR #1919: Use oinspect in CodeMagics._find_edit_target
- PR #1918: don't warn in iptest if deathrow/quarantine are missing
- PR #1917: Fix for %pdef on Python 3
- PR #1913: Fix for #1428
- PR #1911: temporarily skip autoreload tests
- PR #1909: Fix for #1908, use os.path.normcase for safe filename comparisons
- PR #1907: py3compat fixes for %%script and tests
- PR #1906: ofind finds non-unique cell magics
- PR #1845: Fixes to inspection machinery for magics
- PR #1902: Workaround fix for gh-1632; minimal revert of gh-1424
- PR #1900: Cython libs
- PR #1899: add ScriptMagics to class list for generated config
- PR #1898: minimize manpages
- PR #1897: use glob for bad exclusion warning

- PR #1855: %%script and %%file magics
- PR #1870: add %%capture for capturing stdout/err
- PR #1861: Use dvipng to format sympy.Matrix
- PR #1867: Fix 1px margin bouncing of selected menu item.
- PR #1889: Reconnect when the websocket connection closes unexpectedly
- PR #1886: Fix a bug in renaming notebook
- PR #1895: Fix error in test suite with ip.system()
- PR #1762: add *locate* entry points
- PR #1883: Fix vertical offset due to bold/italics, and bad browser fonts.
- PR #1875: re-write columnize, with intermediate step.
- PR #1851: new completer for qtconsole.
- PR #1892: Remove suspicious quotes in interactiveshell.py
- PR #1864: Rmagic exceptions
- PR #1829: [notebook] don't care about leading prct in completion
- PR #1832: Make svg, jpeg and png images resizable in notebook.
- PR #1674: HTML Notebook carriage-return handling, take 2
- PR #1882: Remove importlib dependency which not available in Python 2.6.
- PR #1879: Correct stack depth for variable expansion in !system commands
- PR #1841: [notebook] deduplicate completion results
- PR #1850: Remove args/kwargs handling in TryNext, fix %paste error messages.
- PR #1663: Keep line-endings in ipynb
- PR #1815: Make : invalid in filenames in the Notebook JS code.
- PR #1819: doc: cleanup the parallel psums example a little
- PR #1839: External cleanup
- PR #1782: fix Magic menu in qtconsole, split in groups
- PR #1862: Minor bind_kernel improvements
- PR #1857: Prevent jumping of window to input when output is clicked.
- PR #1856: Fix 1px jumping of cells and menus in Notebook.
- PR #1852: fix chained resubmissions
- PR #1780: Rmagic extension
- PR #1847: add InlineBackend to ConsoleApp class list
- PR #1836: preserve header for resubmitted tasks

- PR #1828: change default extension to .ipy for %save -r
- PR #1800: Reintroduce recall
- PR #1830: lsmagic lists magics in alphabetical order
- PR #1773: Update SymPy profile: SymPy's latex() can now print set and frozenset
- PR #1761: Edited documentation to use IPYTHONDIR in place of ~/.ipython
- PR #1822: aesthetics pass on AsyncResult.display_outputs
- PR #1821: ENTER submits the rename notebook dialog.
- PR #1820: NotebookApp: Make the number of ports to retry user configurable.
- PR #1816: Always use filename as the notebook name.
- PR #1813: Add assert_in method to nose for Python 2.6
- PR #1711: New Tooltip, New Completer and JS Refactor
- PR #1798: a few simple fixes for docs/parallel
- PR #1812: Ensure AsyncResult.display_outputs doesn't display empty streams
- PR #1811: warn on nonexistent exclusions in iptest
- PR #1810: fix for #1809, failing tests in IPython.zmq
- PR #1808: Reposition alternate upload for firefox [need cross browser/OS/language test]
- PR #1742: Check for custom_exceptions only once
- PR #1807: add missing cython exclusion in iptest
- PR #1805: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with m...
- PR #1739: Dashboard improvement (necessary merge of #1658 and #1676 + fix #1492)
- PR #1770: Cython related magic functions
- PR #1707: Accept –gui=<...> switch in IPython qtconsole.
- PR #1797: Fix comment which breaks Emacs syntax highlighting.
- PR #1795: fix %gui magic
- PR #1793: Raise repr limit for strings to 80 characters (from 30).
- PR #1794: don't use XDG path on OS X
- PR #1792: Unicode-aware logger
- PR #1791: update zmqshell magics
- PR #1787: DOC: Remove regression from qt-console docs.
- PR #1758: test_pr, fallback on http if git protocol fail, and SSL errors...
- PR #1748: Fix some tests for Python 3.3
- PR #1755: test for pygments before running qt tests

- PR #1771: Make default value of interactivity passed to run_ast_nodes configurable
- PR #1784: restore loadpy to load
- PR #1768: Update parallel magics
- PR #1779: Tidy up error raising in magic decorators.
- PR #1769: Allow cell mode timeit without setup code.
- PR #1716: Fix for fake filenames in verbose traceback
- PR #1763: [qtconsole] fix append_plain_html -> append_html
- PR #1732: Refactoring of the magics system and implementation of cell magics
- PR #1630: Merge divergent Kernel implementations
- PR #1705: [notebook] Make pager resizable, and remember size...
- PR #1606: Share code for %pycat and %loadpy, make %pycat aware of URLs
- PR #1757: Open IPython notebook hyperlinks in a new window using target=_blank
- PR #1754: Fix typo enconters->encounters
- PR #1753: Clear window title when kernel is restarted
- PR #1449: Fix for bug #735 : Images missing from XML/SVG export
- PR #1743: Tooltip completer js refactor
- PR #1681: add qt config option to clear_on_kernel_restart
- PR #1733: Tooltip completer js refactor
- PR #1727: terminate kernel after embed_kernel tests
- PR #1737: add HistoryManager to ipapp class list
- PR #1686: ENH: Open a notebook from the command line
- PR #1709: fixes #1708, failing test in arg_split on windows
- PR #1718: Use CRegExp trait for regular expressions.
- PR #1729: Catch failure in repr() for %whos
- PR #1726: use eval for command-line args instead of exec
- PR #1724: fix scatter/gather with targets='all'
- PR #1725: add –no-ff to git pull in test_pr
- PR #1721: Tooltip completer js refactor
- PR #1657: Add *wait* optional argument to *hooks.editor*
- PR #1717: Define generic sys.ps{1,2,3}, for use by scripts.
- PR #1691: Finish PR #1446
- PR #1710: update MathJax CDN url for https

- PR #1713: Make autocall regexp's configurable.
- PR #1703: Allow TryNext to have an error message without it affecting the command chain
- PR #1714: minor adjustments to test_pr
- PR #1704: ensure all needed qt parts can be imported before settling for one
- PR #1706: Mark test_push_numpy_nocopy as a known failure for Python 3
- PR #1698: fix tooltip on token with number
- PR #1245: pythonw py3k fixes for issue #1226
- PR #1685: Add script to test pull request
- PR #1693: deprecate IPYTHON_DIR in favor of IPYTHONDIR
- PR #1695: Avoid deprecated warnings from ipython-qtconsole.desktop.
- PR #1694: Add quote to notebook to allow it to load
- PR #1689: Fix sys.path missing '' as first entry in *ipython kernel*.
- PR #1687: import Binary from bson instead of pymongo
- PR #1616: Make IPython.core.display.Image less notebook-centric
- PR #1684: CLN: Remove redundant function definition.
- PR #1670: Point %pastebin to gist
- PR #1669: handle pyout messages in test_message_spec
- PR #1295: add binary-tree engine interconnect example
- PR #1642: Cherry-picked commits from 0.12.1 release
- PR #1659: Handle carriage return characters ("r") in HTML notebook output.
- PR #1656: ensure kernels are cleaned up in embed_kernel tests
- PR #1664: InteractiveShell.run_code: Update docstring.
- PR #1662: Delay flushing softspace until after cell finishes
- PR #1643: handle jpg/jpeg in the qtconsole
- PR #1652: add patch_pyzmq() for backporting a few changes from newer pyzmq
- PR #1650: DOC: moving files with SSH launchers
- PR #1357: add IPython.embed_kernel()
- PR #1640: Finish up embed_kernel
- PR #1651: Remove bundled Itpl module
- PR #1634: incremental improvements to SSH launchers
- PR #1649: move examples/test_embed into examples/tests/embed
- PR #1633: Fix installing extension from local file on Windows

- PR #1645: Exclude UserDict when deep reloading NumPy.
- PR #1637: Removed a ':' which shouldn't have been there
- PR #1631: TST: QApplication doesn't quit early enough with PySide.
- PR #1629: evaluate a few dangling validate_message generators
- PR #1621: clear In[] prompt numbers on "Clear All Output"
- PR #1627: Test the Message Spec
- PR #1624: Fixes for byte-compilation on Python 3
- PR #1615: Add show() method to figure objects.
- PR #1625: Fix deepreload on Python 3
- PR #1620: pyin message now have execution_count
- PR #1457: Update deepreload to use a rewritten knee.py. Fixes dreload(numpy).
- PR #1613: allow map / parallel function for single-engine views
- PR #1609: exit notebook cleanly on SIGINT, SIGTERM
- PR #1607: cleanup sqlitedb temporary db file after tests
- PR #1608: don't rely on timedelta.total_seconds in AsyncResult
- PR #1599: Fix for %run -d on Python 3
- PR #1602: Fix %env magic on Python 3.
- PR #1603: Remove python3 profile
- PR #1604: Exclude IPython.quarantine from installation
- PR #1600: Specify encoding for io.open in notebook_reformat tests
- PR #1605: Small fixes for Animation and Progress notebook
- PR #1529: __all__ feature, improvement to dir2, and tests for both
- PR #1548: add sugar methods/properties to AsyncResult
- PR #1535: Fix pretty printing dispatch
- PR #1399: Use LaTeX to print various built-in types with the SymPy printing extension
- PR #1597: re-enter kernel.eventloop after catching SIGINT
- PR #1490: rename plaintext cell -> raw cell
- PR #1480: Fix %notebook magic, etc. nbformat unicode tests and fixes
- PR #1588: Gtk3 integration with ipython works.
- PR #1595: Examples syntax (avoid errors installing on Python 3)
- PR #1526: Find encoding for Python files
- PR #1594: Fix writing git commit ID to a file on build with Python 3

- PR #1556: shallow-copy DictDB query results
- PR #1502: small changes in response to pyflakes pass
- PR #1445: Don't build sphinx docs for sdists
- PR #1538: store git commit hash in utils._sysinfo instead of hidden data file
- PR #1546: attempt to suppress exceptions in channel threads at shutdown
- PR #1559: update tools/github_stats.py to use GitHub API v3
- PR #1563: clear_output improvements
- PR #1560: remove obsolete discussion of Twisted/trial from testing docs
- PR #1569: BUG: qtconsole – non-standard handling of a and b. [Fixes #1561]
- PR #1573: BUG: Ctrl+C crashes wx pylab kernel in qtconsole.
- PR #1568: fix PR #1567
- PR #1567: Fix: openssh_tunnel did not parse port in *server*
- PR #1565: fix AsyncResult.abort
- PR #1552: use os.getcwdu in NotebookManager
- PR #1541: display_pub flushes stdout/err
- PR #1544: make MultiKernelManager.kernel_manager_class configurable
- PR #1517: Fix indentation bug in IPython/lib/pretty.py
- PR #1519: BUG: Include the name of the exception type in its pretty format.
- PR #1489: Fix zero-copy push
- PR #1477: fix dangling *buffer* in IPython.parallel.util
- PR #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
- PR #1481: BUG: Improve placement of CallTipWidget
- PR #1496: BUG: LBYL when clearing the output history on shutdown.
- PR #1508: fix sorting profiles in clustermanager
- PR #1495: BUG: Fix pretty-printing for overzealous objects
- PR #1472: more general fix for #662
- PR #1483: updated magic_history docstring
- PR #1383: First version of cluster web service.
- PR #1398: fix %tb after SyntaxError
- PR #1440: Fix for failing testsuite when using –with-xml-coverage on windows.
- PR #1419: Add %install_ext magic function.
- PR #1424: Win32 shell interactivity

- PR #1468: Simplify structure of a Job in the TaskScheduler
- PR #1447: 1107 - Tab autocompletion can suggest invalid syntax
- PR #1469: Fix typo in comment (insert space)
- PR #1463: Fix completion when importing modules in the cwd.
- PR #1466: Fix for issue #1437, unfriendly windows qtconsole error handling
- PR #1432: Fix ipython directive
- PR #1465: allow *ipython help subcommand* syntax
- PR #1416: Conditional import of ctypes in inputhook
- PR #1462: expedite parallel tests
- PR #1410: Add javascript library and css stylesheet loading to JS class.
- PR #1448: Fix for #875 Never build unicode error messages
- PR #1458: use eval to uncan References
- PR #1450: load mathjax from CDN via https
- PR #1451: include heading level in JSON
- PR #1444: Fix pyhton -> python typos
- PR #1414: ignore errors in shell.var_expand
- PR #1430: Fix for tornado check for tornado < 1.1.0
- PR #1413: get_home_dir expands symlinks, adjust test accordingly
- PR #1385: updated and prettified magic doc strings
- PR #1406: Browser selection
- PR #1377: Saving non-ascii history
- PR #1402: fix symlinked /home issue for FreeBSD
- PR #1405: Only monkeypatch xunit when the tests are run using it.
- PR #1395: Xunit & KnownFailure
- PR #1396: Fix for %tb magic.
- PR #1386: Jsd3
- PR #1388: Add simple support for running inside a virtualenv
- PR #1391: Improve Hub/Scheduler when no engines are registered
- PR #1369: load header with engine id when engine dies in TaskScheduler
- PR #1353: Save notebook as script using unicode file handle.
- PR #1352: Add '-m mod : run library module as a script' option.
- PR #1363: Fix some minor color/style config issues in the qtconsole

- PR #1371: Adds a quiet keyword to sync_imports
- PR #1387: Fixing Cell menu to update cell type select box.
- PR #1296: Wx gui example: fixes the broken example for *%gui wx*.
- PR #1372: ipcontroller cleans up connection files unless reuse=True
- PR #1374: remove calls to meaningless ZMQStream.on_err
- PR #1370: allow draft76 websockets (Safari)
- PR #1368: Ensure handler patterns are str, not unicode
- PR #1361: Notebook bug fix branch
- PR #1364: avoid jsonlib returning Decimal
- PR #1362: Don't log complete contents of history replies, even in debug
- PR #1347: fix weird magic completion in notebook
- PR #1346: fixups for alternate URL prefix stuff
- PR #1336: crack at making notebook.html use the layout.html template
- PR #1331: RST and heading cells
- PR #1247: fixes a bug causing extra newlines after comments.
- PR #1332: notebook - allow prefixes in URL path.
- PR #1341: Don't attempt to tokenize binary files for tracebacks
- PR #1334: added key handler for control-s to notebook, seems to work pretty well
- PR #1338: Fix see also in docstrings so API docs build
- PR #1335: Notebook toolbar UI
- PR #1299: made notebook.html extend layout.html
- PR #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
- PR #1328: Coverage
- PR #1206: don't preserve fixConsole output in json
- PR #1330: Add linewrapping to text cells (new feature in CodeMirror).
- PR #1309: Inoculate clearcmd extension into %reset functionality
- PR #1327: Updatecm2
- PR #1326: Removing Ace edit capability.
- PR #1325: forgotten selected_cell -> get_selected_cell
- PR #1316: Pass subprocess test runners a suitable location for xunit output
- PR #1303: Updatecm
- PR #1312: minor heartbeat tweaks

- PR #1306: Fix %prun input parsing for escaped characters (closes #1302)
- PR #1301: New "Fix for issue #1202" based on current master.
- PR #1289: Make autoreload extension work on Python 3.
- PR #1288: Don't ask for confirmation when stdin isn't available
- PR #1294: TaskScheduler.hwm default to 1 instead of 0
- PR #1283: HeartMonitor.period should be an Integer
- PR #1264: Aceify
- PR #1284: a fix for GH 1269
- PR #1213: BUG: Minor typo in history_console_widget.py
- PR #1267: add NoDB for non-recording Hub
- PR #1222: allow Reference as callable in map/apply
- PR #1257: use self.kernel_manager_class in qtconsoleapp
- PR #1253: set auto_create flag for notebook apps
- PR #1262: Heartbeat no longer shares the app's Context
- PR #1229: Fix display of SyntaxError in Python 3
- PR #1256: Dewijmoize
- PR #1246: Skip tests that require X, when importing pylab results in RuntimeError.
- PR #1211: serve local files in notebook-dir
- PR #1224: edit text cells on double-click instead of single-click
- PR #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
- PR #1207: fix loadpy duplicating newlines
- PR #1129: Unified setup.py
- PR #1199: Reduce IPython.external.*
- PR #1218: Added -q option to %prun for suppression of the output, along with editing the dochelp string.
- PR #1217: Added -q option to %prun for suppression of the output, along with editing the dochelp string
- PR #1175: core.completer: Clean up excessive and unused code.
- PR #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
- PR #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.

Issues (742):

- #1943: add screenshot and link into releasenotes
- #1570: [notebook] remove 'left panel' references from example.

- #1954: update some example notebooks
- #2048: move _encode_binary to jsonutil.encode_images
- #2050: only add quotes around xunit-file on Windows
- #2047: disable auto-scroll on mozilla
- #1258: Magic %paste error
- #2015: Fixes for %paste with special transformations
- #760: Windows: test runner fails if repo path contains spaces
- #2046: Iptest unicode
- #1939: Namespaces
- #2042: increase auto-scroll threshold to 100 lines
- #2043: move RemoteError import to top-level
- #641: In %magic help, remove duplicate aliases
- #2036: %alias_magic
- #1968: Proposal of icons for .ipynb files
- #825: keyboardinterrupt crashes gtk gui when gtk.set_interactive is not available
- #1971: Remove duplicate magics docs
- #2040: Namespaces for cleaner public APIs
- #2039: ipython parallel import exception
- #2035: Getdefaultencoding test error with sympy 0.7.1_git
- #2037: remove *ipython-qtconsole* gui-script
- #1516: ipython-qtconsole script isn’t installed for Python 2.x
- #1297: “ipython -p sh” is in documentation but doesn’t work
- #2038: add extra clear warning to shell doc
- #1265: please ship unminified js and css sources
- #2029: Ship unminified js
- #1920: Provide an easy way to override the Qt widget used by qtconsole
- #2007: Add custom_control and custom_page_control variables to override the Qt widgets used by qtconsole
- #2009: In %magic help, remove duplicate aliases
- #2033: ipython parallel pushing and pulling recarrays
- #2034: fix&test push/pull recarrays
- #2028: Reduce unhelpful information shown by pinfo

- #1992: Tab completion fails with many spaces in filename
- #1885: handle too old wx
- #2030: check wxPython version in inputhook
- #2024: Make interactive_usage a bit more rst friendly
- #2031: disable ^C^C confirmation on Windows
- #2023: Unicode test failure on OS X
- #2027: match stdin encoding in frontend readline test
- #1901: Windows: parallel test fails assert, leaves 14 python processes alive
- #2025: Fix parallel test on WinXP - wait for resource cleanup.
- #1986: Line magic function *%R* not found. (Rmagic)
- #1712: test failure in ubuntu package daily build
- #1183: 0.12 testsuite failures
- #2016: BUG: test runner fails in Windows if filenames contain spaces.
- #1806: Alternate upload methods in firefox
- #2019: Windows: home directory expansion test fails
- #2020: Fix home path expansion test in Windows.
- #2017: Windows core test error - filename quoting
- #2021: Fix Windows pathname issue in ‘odd encoding’ test.
- #1998: call to nt.assert_true(path._writable_dir(home)) returns false in test_path.py
- #2022: don’t check writability in test for get_home_dir when HOME is undefined
- #1589: Test failures and docs don’t build on Mac OS X Lion
- #1996: frontend test tweaks
- #2011: Notebook server can’t start cluster with hyphen-containing profile name
- #2014: relax profile regex in notebook
- #2013: brew install pyqt
- #2005: Strange output artifacts in footer of notebook
- #2012: Mono cursor offset
- #2004: Clarify generic message spec vs. Python message API in docs
- #2006: Don’t crash when starting notebook server if runnable browser not found
- #2010: notebook: Print a warning (but do not abort) if no webbrowser can be found.
- #2008: pip install virtualenv
- #2003: Wrong case of rmagic in docs

- #2002: Refactor %magic into a lsmagic_docs API function.
- #2000: kernel.js consistency with generic IPython message format.
- #1999: *%magic* help: display line and cell magics in alphabetical order.
- #1635: test_prun_quotes fails on Windows
- #1984: Cannot restart Notebook when using *%%script –bg*
- #1981: Clean BG processes created by %%script on kernel exit
- #1994: Fix RST misformatting.
- #1949: Introduce Notebook Magics
- #1985: Kernels should start in notebook dir when manually specified
- #1980: Notebook should check that –notebook-dir exists
- #1951: minor notebook startup/notebook-dir adjustments
- #1969: tab completion in notebook for paths not triggered
- #1974: Allow path completion on notebook.
- #1964: allow multiple instances of a Magic
- #1960: %page not working
- #1991: fix _ofind attr in %page
- #1982: Shutdown qtconsole problem?
- #1988: check for active frontend in update_restart_checkbox
- #1979: Add support for tox (http://tox.testrun.org/) and Travis CI (http://travis-ci.org/)
- #1989: Parallel: output of %px and %px${suffix} is inconsistent
- #1966: ValueError: packer could not serialize a simple message
- #1987: Notebook: MathJax offline install not recognized
- #1970: dblclick to restore size of images
- #1983: Notebook does not save heading level
- #1978: Notebook names truncating at the first period
- #1553: Limited size of output cells and provide scroll bars for such output cells
- #1825: second attempt at scrolled long output
- #1915: add cell-level metadata
- #1934: Cell/Worksheet metadata
- #1746: Confirm restart (configuration option, and checkbox UI)
- #1790: Commenting function.
- #1767: Tab completion problems with cell magics

- #1944: [qtconsole] take %,%% prefix into account for completion
- #1973: fix another FreeBSD $HOME symlink issue
- #1972: Fix completion of '%tim' in the Qt console
- #1887: Make it easy to resize jpeg/png images back to original size.
- #1967: Fix psums example description in docs
- #1678: ctrl-z clears cell output in notebook when pressed enough times
- #1965: fix for #1678, undo no longer clears cells
- #1952: avoid duplicate "Websockets closed" dialog on ws close
- #1961: UnicodeDecodeError on directory with unicode chars in prompt
- #1963: styling prompt, {color.Normal} excepts
- #1962: Support unicode prompts
- #1959: %page not working on qtconsole for Windows XP 32-bit
- #1955: update to latest version of vim-ipython
- #1945: Add –proc option to %%script
- #1957: fix indentation in kernel.js
- #1956: move import RemoteError after get_exc_info
- #1950: Fix for copy action (Ctrl+C) when there is no pager defined in qtconsole
- #1948: Fix help string for InteractiveShell.ast_node_interactivity
- #1941: script magics cause terminal spam
- #1942: swallow stderr of which in utils.process.find_cmd
- #1833: completer draws slightly too small on Chrome
- #1940: fix completer css on some Chrome versions
- #1938: remove remaining references to deprecated XREP/XREQ names
- #1924: HTML superscripts not shown raised in the notebook
- #1925: Fix styling of superscripts and subscripts. Closes #1924.
- #1461: User notification if notebook saving fails
- #1936: increase duration of save messages
- #1542: %save magic fails in clients without stdin if file already exists
- #1937: add %save -f
- #1572: pyreadline version dependency not correctly checked
- #1935: add version checking to pyreadline import test
- #1849: Octave magics

- #1759: github, merge PR(s) just by number(s)
- #1931: Win py3fixes
- #1646: Meaning of restart parameter in client.shutdown() unclear
- #1933: oinspect.find_file: Additional safety if file cannot be found.
- #1916: %paste doesn't work on py3
- #1932: Fix adding functions to CommandChainDispatcher with equal priority on Py 3
- #1928: Select NoDB by default
- #1923: Add IPython syntax support to the %timeit magic, in line and cell mode
- #1926: Make completer recognize escaped quotes in strings.
- #1929: Ipython-qtconsole (0.12.1) hangs with Python 2.7.3, Windows 7 64 bit
- #1409: [qtconsole] forward delete bring completion into current line
- #1922: py3k compatibility for setupegg.py
- #1598: document that sync_imports() can't handle "import foo as bar"
- #1893: Update Parallel Magics and Exception Display
- #1890: Docstrings for magics that use @magic_arguments are rendered wrong
- #1921: magic_arguments: dedent but otherwise preserve indentation.
- #1919: Use oinspect in CodeMagics._find_edit_target
- #1918: don't warn in iptest if deathrow/quarantine are missing
- #1914: %pdef failing on python3
- #1917: Fix for %pdef on Python 3
- #1428: Failing test that prun does not clobber string escapes
- #1913: Fix for #1428
- #1911: temporarily skip autoreload tests
- #1549: autoreload extension crashes ipython
- #1908: find_file errors on windows
- #1909: Fix for #1908, use os.path.normcase for safe filename comparisons
- #1907: py3compat fixes for %%script and tests
- #1904: %%px? doesn't work, shows info for %px, general cell magic problem
- #1906: ofind finds non-unique cell magics
- #1894: Win64 binary install fails
- #1799: Source file not found for magics
- #1845: Fixes to inspection machinery for magics

- #1774: Some magics seems broken
- #1586: Clean up tight coupling between Notebook, CodeCell and Kernel Javascript objects
- #1632: Win32 shell interactivity apparently broke qtconsole "cd" magic
- #1902: Workaround fix for gh-1632; minimal revert of gh-1424
- #1900: Cython libs
- #1503: Cursor is offset in notebook in Chrome 17 on Linux
- #1426: Qt console doesn't handle the *–gui* flag correctly.
- #1180: Can't start IPython kernel in Spyder
- #581: test IPython.zmq
- #1593: Name embedded in notebook overrides filename
- #1899: add ScriptMagics to class list for generated config
- #1618: generate or minimize manpages
- #1898: minimize manpages
- #1896: Windows: apparently spurious warning 'Excluding nonexistent file' ... test_exampleip
- #1897: use glob for bad exclusion warning
- #1215: updated %quickref to show short-hand for %sc and %sx
- #1855: %%script and %%file magics
- #1863: Ability to silence a cell in the notebook
- #1870: add %%capture for capturing stdout/err
- #1861: Use dvipng to format sympy.Matrix
- #1867: Fix 1px margin bouncing of selected menu item.
- #1889: Reconnect when the websocket connection closes unexpectedly
- #1577: If a notebook loses its network connection WebSockets won't reconnect
- #1886: Fix a bug in renaming notebook
- #1895: Fix error in test suite with ip.system()
- #1762: add *locate* entry points
- #1883: Fix vertical offset due to bold/italics, and bad browser fonts.
- #1875: re-write columnize, with intermediate step.
- #1860: IPython.utils.columnize sometime wrong...
- #1851: new completer for qtconsole.
- #1892: Remove suspicious quotes in interactiveshell.py
- #1854: Class *%hierarchy* and graphiz *%%dot* magics

- #1827: Sending tracebacks over ZMQ should protect against unicode failure
- #1864: Rmagic exceptions
- #1829: [notebook] don't care about leading prct in completion
- #1832: Make svg, jpeg and png images resizable in notebook.
- #1674: HTML Notebook carriage-return handling, take 2
- #1874: cython_magic uses importlib, which doesn't ship with py2.6
- #1882: Remove importlib dependency which not available in Python 2.6.
- #1878: shell access using ! will not fill class or function scope vars
- #1879: Correct stack depth for variable expansion in !system commands
- #1840: New JS completer should merge completions before display
- #1841: [notebook] deduplicate completion results
- #1736: no good error message on missing tkinter and %paste
- #1741: Display message from TryNext error in magic_paste
- #1850: Remove args/kwargs handling in TryNext, fix %paste error messages.
- #1663: Keep line-endings in ipynb
- #1872: Matplotlib window freezes using intreractive plot in qtconsole
- #1869: Improve CodeMagics._find_edit_target
- #1781: Colons in notebook name causes notebook deletion without warning
- #1815: Make : invalid in filenames in the Notebook JS code.
- #1819: doc: cleanup the parallel psums example a little
- #1838: externals cleanup
- #1839: External cleanup
- #1782: fix Magic menu in qtconsole, split in groups
- #1862: Minor bind_kernel improvements
- #1859: kernmagic during console startup
- #1857: Prevent jumping of window to input when output is clicked.
- #1856: Fix 1px jumping of cells and menus in Notebook.
- #1848: task fails with "AssertionError: not enough buffers!" after second resubmit
- #1852: fix chained resubmissions
- #1780: Rmagic extension
- #1853: Fix jumpy notebook behavior
- #1842: task with UnmetDependency error still owned by engine

- #1847: add InlineBackend to ConsoleApp class list
- #1846: Exceptions within multiprocessing crash Ipython notebook kernel
- #1843: Notebook does not exist and permalinks
- #1837: edit magic broken in head
- #1834: resubmitted tasks doesn't have same session name
- #1836: preserve header for resubmitted tasks
- #1776: fix magic menu in qtconsole
- #1828: change default extension to .ipy for %save -r
- #1800: Reintroduce recall
- #1671: __future__ environments
- #1830: lsmagic lists magics in alphabetical order
- #1835: Use Python import in ipython profile config
- #1773: Update SymPy profile: SymPy's latex() can now print set and frozenset
- #1761: Edited documentation to use IPYTHONDIR in place of ~/.ipython
- #1772: notebook autocomplete fail when typing number
- #1822: aesthetics pass on AsyncResult.display_outputs
- #1460: Redirect http to https for notebook
- #1287: Refactor the notebook tab completion/tooltip
- #1596: In rename dialog, <return> should submit
- #1821: ENTER submits the rename notebook dialog.
- #1750: Let the user disable random port selection
- #1820: NotebookApp: Make the number of ports to retry user configurable.
- #1816: Always use filename as the notebook name.
- #1775: assert_in not present on Python 2.6
- #1813: Add assert_in method to nose for Python 2.6
- #1498: Add tooltip keyboard shortcuts
- #1711: New Tooltip, New Completer and JS Refactor
- #1798: a few simple fixes for docs/parallel
- #1818: possible bug with latex / markdown
- #1647: Aborted parallel tasks can't be resubmitted
- #1817: Change behavior of ipython notebook –port=...
- #1738: IPython.embed_kernel issues

- #1610: Basic bold and italic in HTML output cells
- #1576: Start and stop kernels from the notebook dashboard
- #1515: impossible to shutdown notebook kernels
- #1812: Ensure AsyncResult.display_outputs doesn't display empty streams
- #1811: warn on nonexistent exclusions in iptest
- #1809: test suite error in IPython.zmq on windows
- #1810: fix for #1809, failing tests in IPython.zmq
- #1808: Reposition alternate upload for firefox [need cross browser/OS/language test]
- #1742: Check for custom_exceptions only once
- #1802: cythonmagic tests should be skipped if Cython not available
- #1062: warning message in IPython.extensions test
- #1807: add missing cython exclusion in iptest
- #1805: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with m...
- #1803: MPI parallel %px bug
- #1804: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with mingw.
- #1492: Drag target very small if IPython Dashboard has no notebooks
- #1562: Offer a method other than drag-n-drop to upload notebooks
- #1739: Dashboard improvement (necessary merge of #1658 and #1676 + fix #1492)
- #1770: Cython related magic functions
- #1532: qtconsole does not accept –gui switch
- #1707: Accept –gui=<...> switch in IPython qtconsole.
- #1797: Fix comment which breaks Emacs syntax highlighting.
- #1796: %gui magic broken
- #1795: fix %gui magic
- #1788: extreme truncating of return values
- #1793: Raise repr limit for strings to 80 characters (from 30).
- #1794: don't use XDG path on OS X
- #1777: ipython crash on wrong encoding
- #1792: Unicode-aware logger
- #1791: update zmqshell magics
- #1787: DOC: Remove regression from qt-console docs.
- #1785: IPython.utils.tests.test_process.SubProcessTestCase

- #1758: test_pr, fallback on http if git protocol fail, and SSL errors...
- #1786: Make notebook save failures more salient
- #1748: Fix some tests for Python 3.3
- #1755: test for pygments before running qt tests
- #1771: Make default value of interactivity passed to run_ast_nodes configurable
- #1783: part of PR #1606 (loadpy -> load) erased by magic refactoring.
- #1784: restore loadpy to load
- #1768: Update parallel magics
- #1778: string exception in IPython/core/magic.py:232
- #1779: Tidy up error raising in magic decorators.
- #1769: Allow cell mode timeit without setup code.
- #1716: Fix for fake filenames in verbose traceback
- #1763: [qtconsole] fix append_plain_html -> append_html
- #1766: Test failure in IPython.parallel
- #1611: IPEP1: Cell magics and general cleanup of the Magic system
- #1732: Refactoring of the magics system and implementation of cell magics
- #1765: test_pr should clearn PYTHONPATH for the subprocesses
- #1630: Merge divergent Kernel implementations
- #1705: [notebook] Make pager resizable, and remember size...
- #1606: Share code for %pycat and %loadpy, make %pycat aware of URLs
- #1720: Adding interactive inline plotting to notebooks with flot
- #1701: [notebook] Open HTML links in a new window by default
- #1757: Open IPython notebook hyperlinks in a new window using target=_blank
- #1735: Open IPython notebook hyperlinks in a new window using target=_blank
- #1754: Fix typo enconters->encounters
- #1753: Clear window title when kernel is restarted
- #735: Images missing from XML/SVG export (for me)
- #1449: Fix for bug #735 : Images missing from XML/SVG export
- #1752: Reconnect Websocket when it closes unexpectedly
- #1751: Reconnect Websocket when it closes unexpectedly
- #1749: Load MathJax.js using HTTPS when IPython notebook server is HTTPS
- #1743: Tooltip completer js refactor

- #1700: A module for sending custom user messages from the kernel.
- #1745: htmlnotebook: Cursor is off
- #1728: ipython crash with matplotlib during picking
- #1681: add qt config option to clear_on_kernel_restart
- #1733: Tooltip completer js refactor
- #1676: Kernel status/shutdown from dashboard
- #1658: Alternate notebook upload methods
- #1727: terminate kernel after embed_kernel tests
- #1737: add HistoryManager to ipapp class list
- #945: Open a notebook from the command line
- #1686: ENH: Open a notebook from the command line
- #1709: fixes #1708, failing test in arg_split on windows
- #1718: Use CRegExp trait for regular expressions.
- #1729: Catch failure in repr() for %whos
- #1726: use eval for command-line args instead of exec
- #1723: scatter/gather fail with targets='all'
- #1724: fix scatter/gather with targets='all'
- #1725: add –no-ff to git pull in test_pr
- #1722: unicode exception when evaluating expression with non-ascii characters
- #1721: Tooltip completer js refactor
- #1657: Add *wait* optional argument to *hooks.editor*
- #123: Define sys.ps{1,2}
- #1717: Define generic sys.ps{1,2,3}, for use by scripts.
- #1442: cache-size issue in qtconsole
- #1691: Finish PR #1446
- #1446: Fixing Issue #1442
- #1710: update MathJax CDN url for https
- #81: Autocall fails if first function argument begins with "-" or "+
- #1713: Make autocall regexp's configurable.
- #211: paste command not working
- #1703: Allow TryNext to have an error message without it affecting the command chain
- #1714: minor adjustments to test_pr

- #1509: New tooltip for notebook
- #1697: Major refactoring of the Notebook, Kernel and CodeCell JavaScript.
- #788: Progress indicator in the notebook (and perhaps the Qt console)
- #1034: Single process Qt console
- #1557: magic function conflict while using –pylab
- #1476: Pylab figure objects not properly updating
- #1704: ensure all needed qt parts can be imported before settling for one
- #1708: test failure in arg_split on windows
- #1706: Mark test_push_numpy_nocopy as a known failure for Python 3
- #1696: notebook tooltip fail on function with number
- #1698: fix tooltip on token with number
- #1226: Windows GUI only (pythonw) bug for IPython on Python 3.x
- #1245: pythonw py3k fixes for issue #1226
- #1417: Notebook Completer Class
- #1690: [Bogus] Deliberately make a test fail
- #1685: Add script to test pull request
- #1167: Settle on a choice for $IPYTHONDIR
- #1693: deprecate IPYTHON_DIR in favor of IPYTHONDIR
- #1672: ipython-qtconsole.desktop is using a deprecated format
- #1695: Avoid deprecated warnings from ipython-qtconsole.desktop.
- #1694: Add quote to notebook to allow it to load
- #1240: sys.path missing ‘’ as first entry when kernel launched without interface
- #1689: Fix sys.path missing ‘’ as first entry in *ipython kernel*.
- #1683: Parallel controller failing with Pymongo 2.2
- #1687: import Binary from bson instead of pymongo
- #1614: Display Image in Qtconsole
- #1616: Make IPython.core.display.Image less notebook-centric
- #1684: CLN: Remove redundant function definition.
- #1655: Add %open magic command to open editor in non-blocking manner
- #1677: middle-click paste broken in notebook
- #1670: Point %pastebin to gist
- #1667: Test failure in test_message_spec

- #1668: Test failure in IPython.zmq.tests.test_message_spec.test_complete "'pyout' != 'status'"
- #1669: handle pyout messages in test_message_spec
- #1295: add binary-tree engine interconnect example
- #1642: Cherry-picked commits from 0.12.1 release
- #1659: Handle carriage return characters ("r") in HTML notebook output.
- #1313: Figure out MathJax 2 support
- #1653: Test failure in IPython.zmq
- #1656: ensure kernels are cleaned up in embed_kernel tests
- #1666: pip install ipython==dev installs version 0.8 from an old svn repo
- #1664: InteractiveShell.run_code: Update docstring.
- #1512: *print stuff,* should avoid newline
- #1662: Delay flushing softspace until after cell finishes
- #1643: handle jpg/jpeg in the qtconsole
- #966: dreload fails on Windows XP with iPython 0.11 "Unexpected Error"
- #1500: dreload doesn't seem to exclude numpy
- #1520: kernel crash when showing tooltip (?)
- #1652: add patch_pyzmq() for backporting a few changes from newer pyzmq
- #1650: DOC: moving files with SSH launchers
- #1357: add IPython.embed_kernel()
- #1640: Finish up embed_kernel
- #1651: Remove bundled Itpl module
- #1634: incremental improvements to SSH launchers
- #1649: move examples/test_embed into examples/tests/embed
- #1171: Recognise virtualenvs
- #1479: test_extension failing in Windows
- #1633: Fix installing extension from local file on Windows
- #1644: Update copyright date to 2012
- #1636: Test_deepreload breaks pylab irunner tests
- #1645: Exclude UserDict when deep reloading NumPy.
- #1454: make it possible to start engine in 'disabled' mode and 'enable' later
- #1641: Escape code for the current time in PromptManager
- #1638: ipython console clobbers custom sys.path

- #1637: Removed a ':' which shouldn't have been there
- #1536: ipython 0.12 embed shell won't run startup scripts
- #1628: error: QApplication already exists in TestKillRing
- #1631: TST: QApplication doesn't quit early enough with PySide.
- #1629: evaluate a few dangling validate_message generators
- #1621: clear In[] prompt numbers on "Clear All Output"
- #1627: Test the Message Spec
- #1470: SyntaxError on setup.py install with Python 3
- #1624: Fixes for byte-compilation on Python 3
- #1612: pylab=inline fig.show() non-existent in notebook
- #1615: Add show() method to figure objects.
- #1622: deepreload fails on Python 3
- #1625: Fix deepreload on Python 3
- #1626: Failure in new *dreload* tests under Python 3.2
- #1623: iPython / matplotlib Memory error with imshow
- #1619: pyin messages should have execution_count
- #1620: pyin message now have execution_count
- #32: dreload produces spurious traceback when numpy is involved
- #1457: Update deepreload to use a rewritten knee.py. Fixes dreload(numpy).
- #1613: allow map / parallel function for single-engine views
- #1609: exit notebook cleanly on SIGINT, SIGTERM
- #1531: Function keyword completion fails if cursor is in the middle of the complete parentheses
- #1607: cleanup sqlitedb temporary db file after tests
- #1608: don't rely on timedelta.total_seconds in AsyncResult
- #1421: ipython32 %run -d breaks with NameError name 'execfile' is not defined
- #1599: Fix for %run -d on Python 3
- #1201: %env magic fails with Python 3.2
- #1602: Fix %env magic on Python 3.
- #1603: Remove python3 profile
- #1604: Exclude IPython.quarantine from installation
- #1601: Security file is not removed after shutdown by Ctrl+C or kill -INT
- #1600: Specify encoding for io.open in notebook_reformat tests

- #1605: Small fixes for Animation and Progress notebook
- #1452: Bug fix for approval
- #13: Improve robustness and debuggability of test suite
- #70: IPython should prioritize __all__ during tab completion
- #1529: __all__ feature, improvement to dir2, and tests for both
- #1475: Custom namespace for %run
- #1564: calling .abort on AsyncMapResult results in traceback
- #1548: add sugar methods/properties to AsyncResult
- #1535: Fix pretty printing dispatch
- #1522: Discussion: some potential Qt console refactoring
- #1399: Use LaTeX to print various built-in types with the SymPy printing extension
- #1597: re-enter kernel.eventloop after catching SIGINT
- #1490: rename plaintext cell -> raw cell
- #1487: %notebook fails in qtconsole
- #1545: trailing newline not preserved in splitline ipynb
- #1480: Fix %notebook magic, etc. nbformat unicode tests and fixes
- #1588: Gtk3 integration with ipython works.
- #1595: Examples syntax (avoid errors installing on Python 3)
- #1526: Find encoding for Python files
- #1594: Fix writing git commit ID to a file on build with Python 3
- #1556: shallow-copy DictDB query results
- #1499: various pyflakes issues
- #1502: small changes in response to pyflakes pass
- #1445: Don't build sphinx docs for sdists
- #1484: unhide .git_commit_info.ini
- #1538: store git commit hash in utils._sysinfo instead of hidden data file
- #1546: attempt to suppress exceptions in channel threads at shutdown
- #1524: unhide git_commit_info.ini
- #1559: update tools/github_stats.py to use GitHub API v3
- #1563: clear_output improvements
- #1558: Ipython testing documentation still mentions twisted and trial
- #1560: remove obsolete discussion of Twisted/trial from testing docs

- #1561: Qtconsole - nonstandard a and b
- #1569: BUG: qtconsole – non-standard handling of a and b. [Fixes #1561]
- #1574: BUG: Ctrl+C crashes wx pylab kernel in qtconsole
- #1573: BUG: Ctrl+C crashes wx pylab kernel in qtconsole.
- #1590: 'iPython3 qtconsole' doesn't work in Windows 7
- #602: User test the html notebook
- #613: Implement Namespace panel section
- #879: How to handle Javascript output in the notebook
- #1255: figure.show() raises an error with the inline backend
- #1467: Document or bundle a git-integrated facility for stripping VCS-unfriendly binary data
- #1237: Kernel status and logout button overlap
- #1319: Running a cell with ctrl+Enter selects text in cell
- #1571: module member autocomplete should respect __all__
- #1566: ipython3 doesn't run in Win7 with Python 3.2
- #1568: fix PR #1567
- #1567: Fix: openssh_tunnel did not parse port in *server*
- #1565: fix AsyncResult.abort
- #1550: Crash when starting notebook in a non-ascii path
- #1552: use os.getcwdu in NotebookManager
- #1554: wrong behavior of the all function on iterators
- #1541: display_pub flushes stdout/err
- #1539: Asynchrous issue when using clear_display and print x,y,z
- #1544: make MultiKernelManager.kernel_manager_class configurable
- #1494: Untrusted Secure Websocket broken on latest chrome dev
- #1521: only install ipython-qtconsole gui script on Windows
- #1528: Tab completion optionally respects __all__ (+ dir2() cleanup)
- #1527: Making a progress bar work in IPython Notebook
- #1497: __all__ functionality added to dir2(obj)
- #1518: Pretty printing exceptions is broken
- #811: Fixes for ipython unhandeled OSError exception on failure of os.getcwdu()
- #1517: Fix indentation bug in IPython/lib/pretty.py
- #1519: BUG: Include the name of the exception type in its pretty format.

- #1525: A hack for auto-complete numpy recarray
- #1489: Fix zero-copy push
- #1401: numpy arrays cannot be used with View.apply() in Python 3
- #1477: fix dangling *buffer* in IPython.parallel.util
- #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
- #1511: Version comparison error ('2.1.11' < '2.1.4' ==> True)
- #1506: "Fixing" the Notebook scroll to help in visually comparing outputs
- #1481: BUG: Improve placement of CallTipWidget
- #1241: When our debugger class is used standalone *_oh* key errors are thrown
- #676: IPython.embed() from ipython crashes twice on exit
- #1496: BUG: LBYL when clearing the output history on shutdown.
- #1507: python3 notebook: TypeError: unorderable types
- #1508: fix sorting profiles in clustermanager
- #1495: BUG: Fix pretty-printing for overzealous objects
- #1505: SQLite objects created in a thread can only be used in that same thread
- #1482: %history documentation out of date?
- #1501: dreload doesn't seem to exclude numpy
- #1472: more general fix for #662
- #1486: save state of qtconsole
- #1485: add history search to qtconsole
- #1483: updated magic_history docstring
- #1383: First version of cluster web service.
- #482: test_run.test_tclass fails on Windows
- #1398: fix %tb after SyntaxError
- #1478: key function or lambda in sorted function doesn't find global variables
- #1415: handle exit/quit/exit()/quit() variants in zmqconsole
- #1440: Fix for failing testsuite when using –with-xml-coverage on windows.
- #1419: Add %install_ext magic function.
- #1424: Win32 shell interactivity
- #1434: Controller should schedule tasks of multiple clients at the same time
- #1268: notebook %reset magic fails with StdinNotImplementedError
- #1438: from cherrypy import expose fails when running script form parent directory

- #1468: Simplify structure of a Job in the TaskScheduler
- #875: never build unicode error messages
- #1107: Tab autocompletion can suggest invalid syntax
- #1447: 1107 - Tab autocompletion can suggest invalid syntax
- #1469: Fix typo in comment (insert space)
- #1463: Fix completion when importing modules in the cwd.
- #1437: unfriendly error handling with pythonw and ipython-qtconsole
- #1466: Fix for issue #1437, unfriendly windows qtconsole error handling
- #1432: Fix ipython directive
- #1465: allow *ipython help subcommand* syntax
- #1394: Wishlist: Remove hard dependency on ctypes
- #1416: Conditional import of ctypes in inputhook
- #1462: expedite parallel tests
- #1418: Strict mode in javascript
- #1410: Add javascript library and css stylesheet loading to JS class.
- #1427: #922 again
- #1448: Fix for #875 Never build unicode error messages
- #1458: use eval to uncan References
- #1455: Python3 install fails
- #1450: load mathjax from CDN via https
- #1182: Qtconsole, multiwindow
- #1439: Notebook not storing heading celltype information
- #1451: include heading level in JSON
- #1444: Fix pyhton -> python typos
- #1412: Input parsing issue with %prun
- #1414: ignore errors in shell.var_expand
- #1441: (1) Enable IPython.notebook.kernel.execute to publish display_* even it is not called with a code cell and (2) remove empty html element when execute “display_*”
- #1431: Beginner Error: ipython qtconsole
- #1436: “ipython-qtconsole –gui qt” hangs on 64-bit win7
- #1433: websocket connection fails on Chrome
- #1430: Fix for tornado check for tornado < 1.1.0

- #1408: test_get_home_dir_3 failed on Mac OS X
- #1413: get_home_dir expands symlinks, adjust test accordingly
- #1420: fixes #922
- #823: KnownFailure tests appearing as errors
- #1385: updated and prettified magic doc strings
- #1406: Browser selection
- #1411: ipcluster starts 8 engines "successfully" but Client only finds two
- #1375: %history -g -f file encoding issue
- #1377: Saving non-ascii history
- #797: Source introspection needs to be smarter in python 3.2
- #846: Autoreload extension doesn't work with Python 3.2
- #1360: IPython notebook not starting on winXP
- #1407: Qtconsole segfaults on OSX when displaying some pop-up function tooltips
- #1402: fix symlinked /home issue for FreeBSD
- #1403: pyreadline cyclic dependency with pdb++/pdbpp module
- #1405: Only monkeypatch xunit when the tests are run using it.
- #1404: Feature Request: List/Dictionary tab completion
- #1395: Xunit & KnownFailure
- #1396: Fix for %tb magic.
- #1397: Stay or leave message not working, Safari session lost.
- #1389: pylab=inline inoperant through ssh tunnelling?
- #1386: Jsd3
- #1388: Add simple support for running inside a virtualenv
- #826: Add support for creation of parallel task when no engine is running
- #1391: Improve Hub/Scheduler when no engines are registered
- #1369: load header with engine id when engine dies in TaskScheduler
- #1345: notebook can't save unicode as script
- #1353: Save notebook as script using unicode file handle.
- #1352: Add '-m mod : run library module as a script' option.
- #1363: Fix some minor color/style config issues in the qtconsole
- #1371: Adds a quiet keyword to sync_imports
- #1390: Blank screen for notebooks on Safari

- #1387: Fixing Cell menu to update cell type select box.
- #645: Standalone WX GUI support is broken
- #1296: Wx gui example: fixes the broken example for *%gui wx*.
- #1254: typo in notebooklist.js breaks links
- #781: Users should be able to clone a notebook
- #1372: ipcontroller cleans up connection files unless reuse=True
- #1374: remove calls to meaningless ZMQStream.on_err
- #1382: Update RO for Notebook
- #1370: allow draft76 websockets (Safari)
- #1368: Ensure handler patterns are str, not unicode
- #1379: Sage link on website homepage broken
- #1376: FWIW does not work with Chrome 16.0.912.77 Ubuntu 10.10
- #1358: Cannot install ipython on Windows 7 64-bit
- #1367: Ctrl - m t does not toggle output in chrome
- #1359: [sympyprinting] MathJax can't render root{m}{n}
- #1337: Tab in the notebook after *(* should not indent, only give a tooltip
- #1339: Notebook printing broken
- #1344: Ctrl + M + L does not toggle line numbering in htmlnotebook
- #1348: Ctrl + M + M does not switch to markdown cell
- #1361: Notebook bug fix branch
- #1364: avoid jsonlib returning Decimal
- #1362: Don't log complete contents of history replies, even in debug
- #888: ReST support in notebooks
- #1205: notebook stores HTML escaped text in the file
- #1351: add IPython.embed_kernel()
- #1243: magic commands without % are not completed properly in htmlnotebook
- #1347: fix weird magic completion in notebook
- #1355: notebook.html extends layout.html now
- #1354: min and max in the notebook
- #1346: fixups for alternate URL prefix stuff
- #1336: crack at making notebook.html use the layout.html template
- #1331: RST and heading cells

- #1350: Add '-m mod : run library module as a script' option
- #1247: fixes a bug causing extra newlines after comments.
- #1329: add base_url to notebook configuration options
- #1332: notebook - allow prefixes in URL path.
- #1317: Very slow traceback construction from Cython extension
- #1341: Don't attempt to tokenize binary files for tracebacks
- #1300: Cell Input collapse
- #1334: added key handler for control-s to notebook, seems to work pretty well
- #1338: Fix see also in docstrings so API docs build
- #1335: Notebook toolbar UI
- #1299: made notebook.html extend layout.html
- #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
- #873: ReST support in notebook frontend
- #1139: Notebook webkit notification
- #1314: Insertcell
- #1328: Coverage
- #1206: don't preserve fixConsole output in json
- #1330: Add linewrapping to text cells (new feature in CodeMirror).
- #1309: Inoculate clearcmd extension into %reset functionality
- #1327: Updatecm2
- #1326: Removing Ace edit capability.
- #1325: forgotten selected_cell -> get_selected_cell
- #1316: Pass subprocess test runners a suitable location for xunit output
- #1315: Collect results from subprocess runners and spit out Xunit XML output.
- #1233: Update CodeMirror to the latest version
- #1234: Refactor how the notebook focuses cells
- #1235: After upgrading CodeMirror check the status of some bugs
- #1236: Review how select is called when notebook cells are inserted
- #1303: Updatecm
- #1311: Fixing CM related indentation problems.
- #1304: controller/server load can disrupt heartbeat
- #1312: minor heartbeat tweaks

- #1302: Input parsing with %prun clobbers escapes
- #1306: Fix %prun input parsing for escaped characters (closes #1302)
- #1251: IPython-0.12 can't import map module on Python 3.1
- #1202: Pyreadline install exclusion for 64 bit windows no longer required, version dependency not correctly specified.
- #1301: New "Fix for issue #1202" based on current master.
- #1242: changed key map name to match changes to python mode
- #1203: Fix for issue #1202
- #1289: Make autoreload extension work on Python 3.
- #1263: Different 'C-x' for shortcut, 'C-m c' not toCodeCell anymore
- #1259: Replace "from (.|..) import" with absolute imports.
- #1278: took a crack at making notebook.html extend layout.html
- #1210: Add 'quiet' option to suppress screen output during %prun calls, edited dochelp
- #1288: Don't ask for confirmation when stdin isn't available
- #1290: Cell-level cut & paste overwrites multiple cells
- #1291: Minor, but important fixes to cut/copy/paste.
- #1293: TaskScheduler.hwm default value
- #1294: TaskScheduler.hwm default to 1 instead of 0
- #1281: in Hub: registration_timeout must be an integer, but heartmonitor.period is CFloat
- #1283: HeartMonitor.period should be an Integer
- #1162: Allow merge/split adjacent cells in notebook
- #1264: Aceify
- #1261: Mergesplit
- #1269: Another strange input handling error
- #1284: a fix for GH 1269
- #1232: Dead kernel loop
- #1279: ImportError: cannot import name S1 (from logging)
- #1276: notebook menu item to send a KeyboardInterrupt to the kernel
- #1213: BUG: Minor typo in history_console_widget.py
- #1248: IPython notebook doesn't work with lastest version of tornado
- #1267: add NoDB for non-recording Hub
- #1222: allow Reference as callable in map/apply

- #1257: use self.kernel_manager_class in qtconsoleapp
- #1220: Open a new notebook while connecting to an existing kernel (opened by qtconsole or terminal or standalone)
- #1253: set auto_create flag for notebook apps
- #1260: heartbeat failure on long gil-holding operation
- #1262: Heartbeat no longer shares the app's Context
- #1225: SyntaxError display broken in Python 3
- #1229: Fix display of SyntaxError in Python 3
- #1256: Dewijmoize
- #1246: Skip tests that require X, when importing pylab results in RuntimeError.
- #1250: Wijmoize
- #1244: can not imput chinese word "" , exit right now
- #1194: Adding Opera 11 as a compatible browser for ipython notebook
- #1198: Kernel Has Died error in Notebook
- #1211: serve local files in notebook-dir
- #1224: edit text cells on double-click instead of single-click
- #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
- #1207: fix loadpy duplicating newlines
- #1060: Always save the .py file to disk next to the .ipynb
- #1066: execute cell in place should preserve the current insertion-point in the notebook
- #1141: "In" numbers are not invalidated when restarting kernel
- #1231: pip on OSX tries to install files in /System directory.
- #1129: Unified setup.py
- #1199: Reduce IPython.external.*
- #1219: Make all the static files path absolute.
- #1218: Added -q option to %prun for suppression of the output, along with editing the dochelp string.
- #1217: Added -q option to %prun for suppression of the output, along with editing the dochelp string
- #1216: Pdb tab completion does not work in QtConsole
- #1197: Interactive shell trying to: from ... import history
- #1175: core.completer: Clean up excessive and unused code.
- #1208: should dv.sync_import print failed imports ?
- #1186: payloadpage.py not used by qtconsole

- #1204: double newline from %loadpy in python notebook (at least on mac)
- #1192: Invalid JSON data
- #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
- #1189: Right justify of 'in' prompt in variable prompt size configurations
- #1185: ipython console not work proper with stdout...
- #1191: profile/startup files not executed with "notebook"
- #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.
- #1174: Remove %install_default_config and %install_profiles

2.6 0.12 Series

2.6.1 Release 0.12.1

IPython 0.12.1 is a bugfix release of 0.12, pulling only bugfixes and minor cleanup from 0.13, timed for the Ubuntu 12.04 LTS release.

See the *list of fixed issues* for specific backported issues.

2.6.2 Release 0.12

IPython 0.12 contains several major new features, as well as a large amount of bug and regression fixes. The 0.11 release brought with it a lot of new functionality and major refactorings of the codebase; by and large this has proven to be a success as the number of contributions to the project has increased dramatically, proving that the code is now much more approachable. But in the refactoring inevitably some bugs were introduced, and we have also squashed many of those as well as recovered some functionality that had been temporarily disabled due to the API changes.

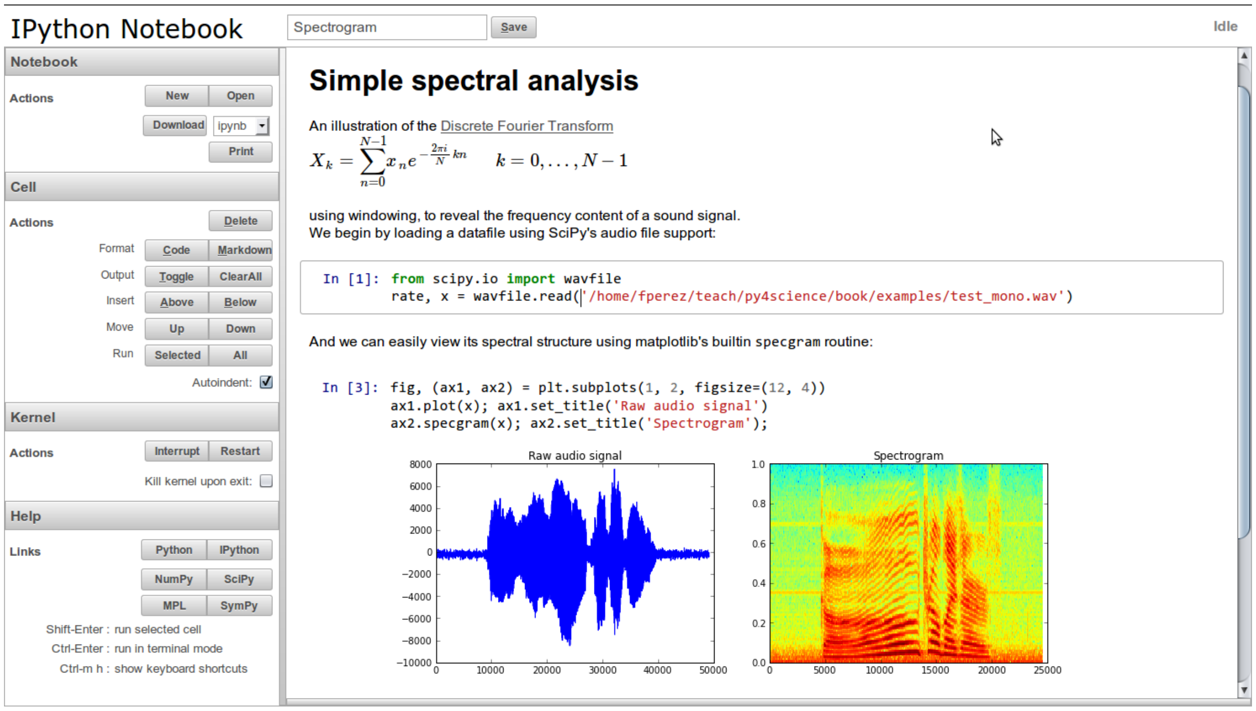
The following major new features appear in this version.

An interactive browser-based Notebook with rich media support

A powerful new interface puts IPython in your browser. You can start it with the command `ipython notebook`:

This new interface maintains all the features of IPython you are used to, as it is a new client that communicates with the same IPython kernels used by the terminal and Qt console. But the web notebook provides for a different workflow where you can integrate, along with code execution, also text, mathematical expressions, graphics, video, and virtually any content that a modern browser is capable of displaying.

You can save your work sessions as documents that retain all these elements and which can be version controlled, emailed to colleagues or saved as HTML or PDF files for printing or publishing statically on the web. The internal storage format is a JSON file that can be easily manipulated for manual exporting to other formats.



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

using windowing, to reveal the frequency content of a sound signal.
We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('/home/fperez/teach/py4science/book/examples/test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [3]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

Fig. 2.3: The new IPython notebook showing text, mathematical expressions in LaTeX, code, results and embedded figures created with Matplotlib.

This Notebook is a major milestone for IPython, as for years we have tried to build this kind of system. We were inspired originally by the excellent implementation in Mathematica, we made a number of attempts using older technologies in earlier Summer of Code projects in 2005 (both students and Robert Kern developed early prototypes), and in recent years we have seen the excellent implementation offered by the *Sage <http://sagemath.org>* system. But we continued to work on something that would be consistent with the rest of IPython's design, and it is clear now that the effort was worth it: based on the ZeroMQ communications architecture introduced in version 0.11, the notebook can now retain 100% of the features of the real IPython. But it can also provide the rich media support and high quality Javascript libraries that were not available in browsers even one or two years ago (such as high-quality mathematical rendering or built-in video).

The notebook has too many useful and important features to describe in these release notes; our documentation now contains a directory called `examples/notebooks` with several notebooks that illustrate various aspects of the system. You should start by reading those named `00_notebook_tour.ipynb` and `01_notebook_introduction.ipynb` first, and then can proceed to read the others in any order you want.

To start the notebook server, go to a directory containing the notebooks you want to open (or where you want to create new ones) and type:

```
ipython notebook
```

You can see all the relevant options with:

```
ipython notebook --help
ipython notebook --help-all   # even more
```

and just like the Qt console, you can start the notebook server with pylab support by using:

```
ipython notebook --pylab
```

for floating matplotlib windows or:

```
ipython notebook --pylab inline
```

for plotting support with automatically inlined figures. Note that it is now possible also to activate pylab support at runtime via `%pylab`, so you do not need to make this decision when starting the server.

See *the Notebook docs* for technical details.

Two-process terminal console

Based on the same architecture as the notebook and the Qt console, we also have now a terminal-based console that can connect to an external IPython kernel (the same kernels used by the Qt console or the notebook, in fact). While this client behaves almost identically to the usual IPython terminal application, this capability can be very useful to attach an interactive console to an existing kernel that was started externally. It lets you use the interactive `%debug` facilities in a notebook, for example (the web browser can't interact directly with the debugger) or debug a third-party code where you may have embedded an IPython kernel.

This is also something that we have wanted for a long time, and which is a culmination (as a team effort) of the work started last year during the 2010 Google Summer of Code project.

Tabbed QtConsole

The QtConsole now supports starting multiple kernels in tabs, and has a menubar, so it looks and behaves more like a real application. Keyboard enthusiasts can disable the menubar with ctrl-shift-M (PR #887).

Full Python 3 compatibility

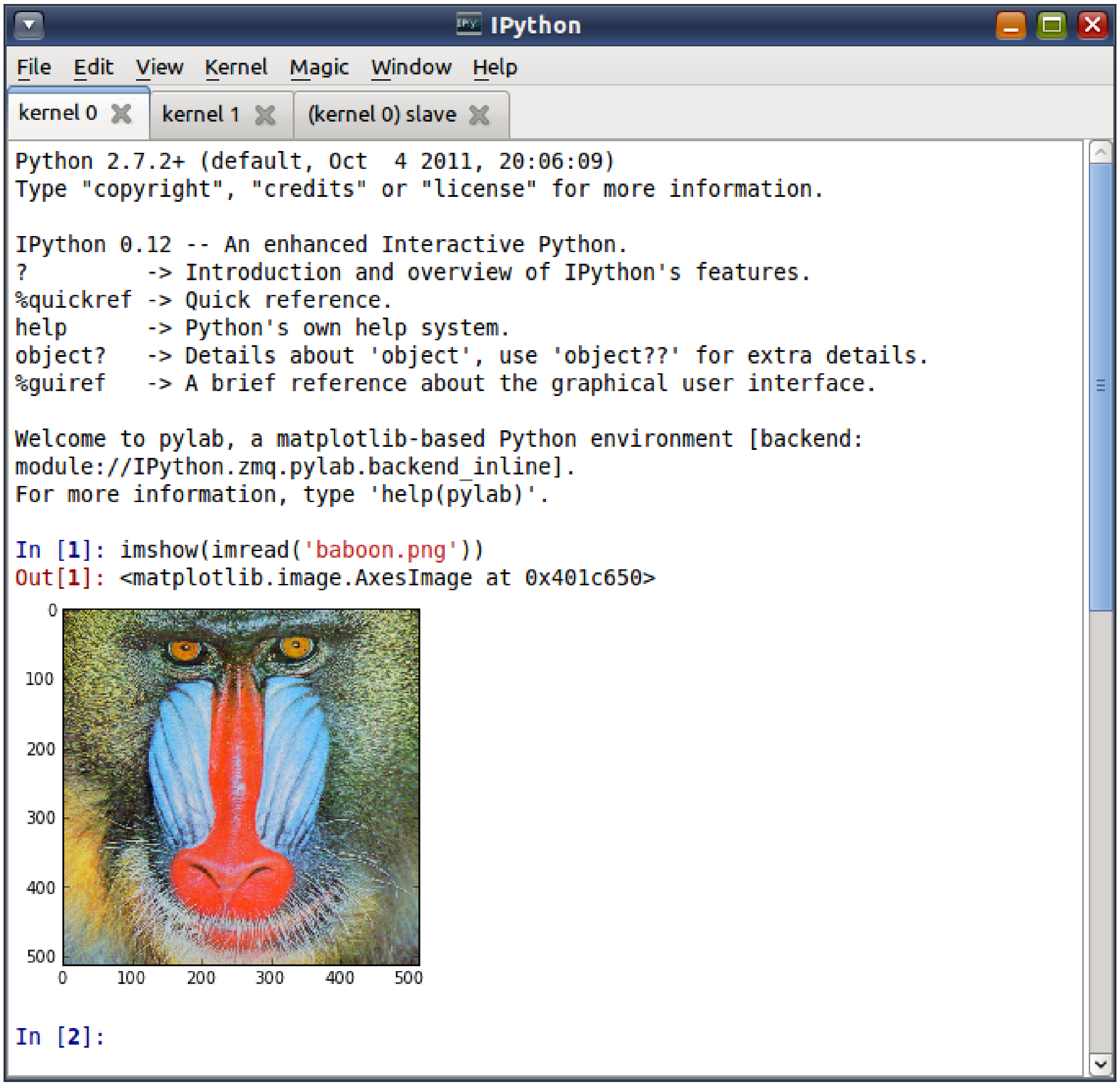
IPython can now be installed from a single codebase on Python 2 and Python 3. The installation process for Python 3 automatically runs 2to3. The same 'default' profile is now used for Python 2 and 3 (the previous version had a separate 'python3' profile).

Standalone Kernel

The `ipython kernel` subcommand has been added, to allow starting a standalone kernel, that can be used with various frontends. You can then later connect a Qt console or a terminal console to this kernel by typing e.g.:

```
ipython qtconsole --existing
```

if it's the only one running, or by passing explicitly the connection parameters (printed by the kernel at startup).



```
Python 2.7.2+ (default, Oct  4 2011, 20:06:09)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- 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.
%guiref   -> A brief reference about the graphical user interface.

Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.

In [1]: imshow(imread('baboon.png'))
Out[1]: <matplotlib.image.AxesImage at 0x401c650>

In [2]:
```

Fig. 2.4: The improved Qt console for IPython, now with tabs to control multiple kernels and full menu support.

PyPy support

The terminal interface to IPython now runs under PyPy. We will continue to monitor PyPy's progress, and hopefully before long at least we'll be able to also run the notebook. The Qt console may take longer, as Qt is a very complex set of bindings to a huge C++ library, and that is currently the area where PyPy still lags most behind. But for everyday interactive use at the terminal, with this release and PyPy 1.7, things seem to work quite well from our admittedly limited testing.

Other important new features

- **SSH Tunnels**: In 0.11, the `IPython.parallel` Client could tunnel its connections to the Controller via ssh. Now, the QtConsole *supports* ssh tunneling, as do parallel engines.
- **relaxed command-line parsing**: 0.11 was released with overly-strict command-line parsing, preventing the ability to specify arguments with spaces, e.g. `ipython --pylab qt` or `ipython -c "print 'hi'"`. This has been fixed, by using argparse. The new parsing is a strict superset of 0.11, so any commands in 0.11 should still work in 0.12.
- **HistoryAccessor**: The `HistoryManager` class for interacting with your IPython SQLite history database has been split, adding a parent `HistoryAccessor` class, so that users can write code to access and search their IPython history without being in an IPython session (PR #824).
- **kernel %gui and %pylab**: The `%gui` and `%pylab` magics have been restored to the IPython kernel (e.g. in the qtconsole or notebook). This allows activation of pylab-mode, or eventloop integration after starting the kernel, which was unavailable in 0.11. Unlike in the terminal, this can be set only once, and cannot be changed.
- **%config**: A new `%config` magic has been added, giving easy access to the IPython configuration system at runtime (PR #923).
- **Multiline History**: Multiline readline history has been restored to the Terminal frontend by default (PR #838).
- **%store**: The `%store` magic from earlier versions has been updated and re-enabled (*storemagic*; PR #1029). To autorestore stored variables on startup, specify `c.StoreMagic.autorestore = True` in `ipython_config.py`.

Major Bugs fixed

In this cycle, we have *closed over 500 issues*, but a few major ones merit special mention:

- Simple configuration errors should no longer crash IPython. In 0.11, errors in config files, as well as invalid trait values, could crash IPython. Now, such errors are reported, and help is displayed.
- Certain SyntaxErrors no longer crash IPython (e.g. just typing keywords, such as `return`, `break`, etc.). See #704.
- IPython path utils, such as `get_ipython_dir()` now check for write permissions, so IPython should function on systems where the default path resolution might point to a read-only location, such as `HOMESHARE` on Windows (#669).

- `raw_input()` now works in the kernel when multiple frontends are in use. The request will be sent to the frontend that made the request, and an exception is raised if that frontend does not support stdin requests (e.g. the notebook) (#673).
- `zmq` version detection no longer uses simple lexicographical comparison to check minimum version, which prevents 0.11 from working with pyzmq-2.1.10 (PR #758).
- A bug in PySide < 1.0.7 caused crashes on OSX when tooltips were shown (#711). these tooltips are now disabled on old PySide (PR #963).
- IPython no longer crashes when started on recent versions of Python 3 in Windows (#737).
- Instances of classes defined interactively can now be pickled (#29; PR #648). Note that pickling saves a reference to the class definition, so unpickling the instances will only work where the class has been defined.

Backwards incompatible changes

- IPython connection information is no longer specified via ip/port directly, rather via json connection files. These files are stored in the security directory, and enable us to turn on HMAC message authentication by default, significantly improving the security of kernels. Various utility functions have been added to `IPython.lib.kernel`, for easier connecting to existing kernels.
- `KernelManager` now has one ip, and several port traits, rather than several ip/port pair `_addr` traits. This better matches the rest of the code, where the ip cannot not be set separately for each channel.
- Custom prompts are now configured using a new class, `PromptManager`, which has traits for `in_template`, `in2_template` (the `...:` continuation prompt), `out_template` and `rewrite_template`. This uses Python's string formatting system, so you can use `{time}` and `{cwd}`, although we have preserved the abbreviations from previous versions, e.g. `\#` (prompt number) and `\w` (working directory). For the list of available fields, refer to the source of `IPython/core/prompts.py`.
- The class inheritance of the Launchers in `IPython.parallel.apps.launcher` used by ipcluster has changed, so that trait names are more consistent across batch systems. This may require a few renames in your config files, if you customized the command-line args for launching controllers and engines. The configurable names have also been changed to be clearer that they point to class names, and can now be specified by name only, rather than requiring the full import path of each class, e.g.:

  ```
  IPClusterEngines.engine_launcher = 'IPython.parallel.apps.launcher.MPIExecEngineSetLau
  IPClusterStart.controller_launcher = 'IPython.parallel.apps.launcher.SSHControllerLaun
  ```

  would now be specified as:

  ```
  IPClusterEngines.engine_launcher_class = 'MPI'
  IPClusterStart.controller_launcher_class = 'SSH'
  ```

  The full path will still work, and is necessary for using custom launchers not in IPython's launcher module.

Further, MPIExec launcher names are now prefixed with just MPI, to better match other batch launchers, and be generally more intuitive. The MPIExec names are deprecated, but continue to work.

- For embedding a shell, note that the parameters `user_global_ns` and `global_ns` have been deprectated in favour of `user_module` and `module` respsectively. The new parameters expect a module-like object, rather than a namespace dict. The old parameters remain for backwards compatibility, although `user_global_ns` is now ignored. The `user_ns` parameter works the same way as before, and calling `embed()` with no arguments still works as before.

Development summary and credits

The previous version (IPython 0.11) was released on July 31 2011, so this release cycle was roughly 4 1/2 months long, we closed a total of 515 issues, 257 pull requests and 258 regular issues (a *detailed list* is available).

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 had commits from the following 45 contributors, a mix of new and regular names (in alphabetical order by first name):

- Alcides <alcides-at-do-not-span-me.com>
- Ben Edwards <bedwards-at-cs.unm.edu>
- Benjamin Ragan-Kelley <benjaminrk-at-gmail.com>
- Benjamin Thyreau <benjamin.thyreau-at-gmail.com>
- Bernardo B. Marques <bernardo.fire-at-gmail.com>
- Bernard Paulus <bprecyclebin-at-gmail.com>
- Bradley M. Froehle <brad.froehle-at-gmail.com>
- Brian E. Granger <ellisonbg-at-gmail.com>
- Christian Boos <cboos-at-bct-technology.com>
- Daniel Velkov <danielv-at-mylife.com>
- Erik Tollerud <erik.tollerud-at-gmail.com>
- Evan Patterson <epatters-at-enthought.com>
- Felix Werner <Felix.Werner-at-kit.edu>
- Fernando Perez <Fernando.Perez-at-berkeley.edu>
- Gabriel <g2p.code-at-gmail.com>
- Grahame Bowland <grahame-at-angrygoats.net>
- Hannes Schulz <schulz-at-ais.uni-bonn.de>
- Jens Hedegaard Nielsen <jenshnielsen-at-gmail.com>
- Jonathan March <jmarch-at-enthought.com>

- Jörgen Stenarson <jorgen.stenarson-at-bostream.nu>
- Julian Taylor <jtaylor.debian-at-googlemail.com>
- Kefu Chai <tchaikov-at-gmail.com>
- macgyver <neil.rabinowitz-at-merton.ox.ac.uk>
- Matt Cottingham <matt.cottingham-at-gmail.com>
- Matthew Brett <matthew.brett-at-gmail.com>
- Matthias BUSSONNIER <bussonniermatthias-at-gmail.com>
- Michael Droettboom <mdboom-at-gmail.com>
- Nicolas Rougier <Nicolas.Rougier-at-inria.fr>
- Olivier Verdier <olivier.verdier-at-gmail.com>
- Omar Andres Zapata Mesa <andresete.chaos-at-gmail.com>
- Pablo Winant <pablo.winant-at-gmail.com>
- Paul Ivanov <pivanov314-at-gmail.com>
- Pauli Virtanen <pav-at-iki.fi>
- Pete Aykroyd <aykroyd-at-gmail.com>
- Prabhu Ramachandran <prabhu-at-enthought.com>
- Puneeth Chaganti <punchagan-at-gmail.com>
- Robert Kern <robert.kern-at-gmail.com>
- Satrajit Ghosh <satra-at-mit.edu>
- Stefan van der Walt <stefan-at-sun.ac.za>
- Szabolcs Horvát <szhorvat-at-gmail.com>
- Thomas Kluyver <takowl-at-gmail.com>
- Thomas Spura <thomas.spura-at-gmail.com>
- Timo Paulssen <timonator-at-perpetuum-immobile.de>
- Valentin Haenel <valentin.haenel-at-gmx.de>
- Yaroslav Halchenko <debian-at-onerussian.com>

Note: This list was generated with the output of `git log rel-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.

2.7 Issues closed in the 0.12 development cycle

2.7.1 Issues closed in 0.12.1

GitHub stats for bugfix release 0.12.1 (12/28/2011-04/16/2012), backporting pull requests from 0.13.

We closed a total of 71 issues: 44 pull requests and 27 issues; this is the full list (generated with the script *tools/github_stats.py*).

This list is automatically generated, and may be incomplete:

Pull Requests (44):

- PR #1175: core.completer: Clean up excessive and unused code.
- PR #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
- PR #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.
- PR #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
- PR #1206: don't preserve fixConsole output in json
- PR #1207: fix loadpy duplicating newlines
- PR #1213: BUG: Minor typo in history_console_widget.py
- PR #1218: Added -q option to %prun for suppression of the output, along with editing the dochelp string.
- PR #1222: allow Reference as callable in map/apply
- PR #1229: Fix display of SyntaxError in Python 3
- PR #1246: Skip tests that require X, when importing pylab results in RuntimeError.
- PR #1253: set auto_create flag for notebook apps
- PR #1257: use self.kernel_manager_class in qtconsoleapp
- PR #1262: Heartbeat no longer shares the app's Context
- PR #1283: HeartMonitor.period should be an Integer
- PR #1284: a fix for GH 1269
- PR #1289: Make autoreload extension work on Python 3.
- PR #1306: Fix %prun input parsing for escaped characters (closes #1302)
- PR #1312: minor heartbeat tweaks
- PR #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
- PR #1341: Don't attempt to tokenize binary files for tracebacks
- PR #1353: Save notebook as script using unicode file handle.
- PR #1363: Fix some minor color/style config issues in the qtconsole

- PR #1364: avoid jsonlib returning Decimal
- PR #1369: load header with engine id when engine dies in TaskScheduler
- PR #1370: allow draft76 websockets (Safari)
- PR #1374: remove calls to meaningless ZMQStream.on_err
- PR #1377: Saving non-ascii history
- PR #1396: Fix for %tb magic.
- PR #1402: fix symlinked /home issue for FreeBSD
- PR #1413: get_home_dir expands symlinks, adjust test accordingly
- PR #1414: ignore errors in shell.var_expand
- PR #1430: Fix for tornado check for tornado < 1.1.0
- PR #1445: Don't build sphinx docs for sdists
- PR #1463: Fix completion when importing modules in the cwd.
- PR #1477: fix dangling *buffer* in IPython.parallel.util
- PR #1495: BUG: Fix pretty-printing for overzealous objects
- PR #1496: BUG: LBYL when clearing the output history on shutdown.
- PR #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
- PR #1517: Fix indentation bug in IPython/lib/pretty.py
- PR #1538: store git commit hash in utils._sysinfo instead of hidden data file
- PR #1599: Fix for %run -d in Python 3
- PR #1602: Fix %env for Python 3
- PR #1607: cleanup sqlitedb temporary db file after tests

Issues (27):

- #676: IPython.embed() from ipython crashes twice on exit
- #846: Autoreload extension doesn't work with Python 3.2
- #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
- #1191: profile/startup files not executed with "notebook"
- #1197: Interactive shell trying to: from ... import history
- #1198: Kernel Has Died error in Notebook
- #1201: %env magic fails with Python 3.2
- #1204: double newline from %loadpy in python notebook (at least on mac)
- #1208: should dv.sync_import print failed imports ?
- #1225: SyntaxError display broken in Python 3

- #1232: Dead kernel loop
- #1241: When our debugger class is used standalone *_oh* key errors are thrown
- #1254: typo in notebooklist.js breaks links
- #1260: heartbeat failure on long gil-holding operation
- #1268: notebook %reset magic fails with StdinNotImplementedError
- #1269: Another strange input handling error
- #1281: in Hub: registration_timeout must be an integer, but heartmonitor.period is CFloat
- #1302: Input parsing with %prun clobbers escapes
- #1304: controller/server load can disrupt heartbeat
- #1317: Very slow traceback construction from Cython extension
- #1345: notebook can't save unicode as script
- #1375: %history -g -f file encoding issue
- #1401: numpy arrays cannot be used with View.apply() in Python 3
- #1408: test_get_home_dir_3 failed on Mac OS X
- #1412: Input parsing issue with %prun
- #1421: ipython32 %run -d breaks with NameError name 'execfile' is not defined
- #1484: unhide .git_commit_info.ini

2.7.2 Issues closed in 0.12

In this cycle, from August 1 to December 28 2011, we closed a total of 515 issues, 257 pull requests and 258 regular issues; this is the full list (generated with the script *tools/github_stats.py*).

Pull requests (257):

- 1174: Remove %install_default_config and %install_profiles
- 1178: Correct string type casting in pinfo.
- 1096: Show class init and call tooltips in notebook
- 1176: Modifications to profile list
- 1173: don't load gui/pylab in console frontend
- 1168: Add –script flag as shorthand for notebook save_script option.
- 1165: encode image_tag as utf8 in [x]html export
- 1161: Allow %loadpy to load remote URLs that don't end in .py
- 1158: Add coding header when notebook exported to .py file.
- 1160: don't ignore ctrl-C during *%gui qt*

- 1159: Add encoding header to Python files downloaded from notebooks.
- 1155: minor post-execute fixes (#1154)
- 1153: Pager tearing bug
- 1152: Add support for displaying maptlotlib axes directly.
- 1079: Login/out button cleanups
- 1151: allow access to user_ns in prompt_manager
- 1120: updated vim-ipython (pending)
- 1150: BUG: Scrolling pager in vsplit on Mac OSX tears.
- 1149: #1148 (win32 arg_split)
- 1147: Put qtconsole forground when launching
- 1146: allow saving notebook.py next to notebook.ipynb
- 1128: fix pylab StartMenu item
- 1140: Namespaces for embedding
- 1132: [notebook] read-only: disable name field
- 1125: notebook : update logo
- 1135: allow customized template and static file paths for the notebook web app
- 1122: BUG: Issue #755 qt IPythonWidget.execute_file fails if filename contains...
- 1137: rename MPIExecLaunchers to MPILaunchers
- 1130: optionally ignore shlex's ValueError in arg_split
- 1116: Shlex unicode
- 1073: Storemagic plugin
- 1143: Add post_install script to create start menu entries in Python 3
- 1138: Fix tests to work when ~/.config/ipython contains a symlink.
- 1121: Don't transform function calls on IPyAutocall objects
- 1118: protect CRLF from carriage-return action
- 1105: Fix for prompts containing newlines.
- 1126: Totally remove pager when read only (notebook)
- 1091: qtconsole : allow copy with shortcut in pager
- 1114: fix magics history in two-process ipython console
- 1113: Fixing #1112 removing failing asserts for test_carriage_return and **test_**...
- 1089: Support carriage return ('r') and beep ('b') characters in the qtconsole
- 1108: Completer usability 2 (rebased of pr #1082)

- 864: Two-process terminal frontend (ipython core branch)
- 1082: usability and cross browser compat for completer
- 1053: minor improvements to text placement in qtconsole
- 1106: Fix display of errors in compiled code on Python 3
- 1077: allow the notebook to run without MathJax
- 1072: If object has a getdoc() method, override its normal docstring.
- 1059: Switch to simple *__IPYTHON__* global
- 1070: Execution count after SyntaxError
- 1098: notebook: config section UI
- 1101: workaround spawnb missing from pexpect.__all__
- 1097: typo, should fix #1095
- 1099: qtconsole export xhtml/utf8
- 1083: Prompts
- 1081: Fix wildcard search for updated namespaces
- 1084: write busy in notebook window title...
- 1078: PromptManager fixes
- 1064: Win32 shlex
- 1069: As you type completer, fix on Firefox
- 1039: Base of an as you type completer.
- 1065: Qtconsole fix racecondition
- 507: Prompt manager
- 1056: Warning in code. qtconsole ssh -X
- 1036: Clean up javascript based on js2-mode feedback.
- 1052: Pylab fix
- 648: Usermod
- 969: Pexpect-u
- 1007: Fix paste/cpaste bug and refactor/cleanup that code a lot.
- 506: make ENTER on a previous input field replace current input buffer
- 1040: json/jsonapi cleanup
- 1042: fix firefox (windows) break line on empty prompt number
- 1015: emacs freezes when tab is hit in ipython with latest python-mode
- 1023: flush stdout/stderr at the end of kernel init

- 956: Generate “All magics...” menu live
- 1038: Notebook: don’t change cell when selecting code using shift+up/down.
- 987: Add Tooltip to notebook.
- 1028: Cleaner minimum version comparison
- 998: defer to stdlib for path.get_home_dir()
- 1033: update copyright to 2011/20xx-2011
- 1032: Intercept <esc> avoid closing websocket on Firefox
- 1030: use pyzmq tools where appropriate
- 1029: Restore pspersistence, including %store magic, as an extension.
- 1025: Dollar escape
- 999: Fix issue #880 - more useful message to user when %paste fails
- 938: changes to get ipython.el to work with the latest python-mode.el
- 1012: Add logout button.
- 1020: Dollar formatter for ! shell calls
- 1019: Use repr() to make quoted strings
- 1008: don’t use crash_handler by default
- 1003: Drop consecutive duplicates when refilling readline history
- 997: don’t unregister interrupted post-exec functions
- 996: add Integer traitlet
- 1016: Fix password hashing for Python 3
- 1014: escape minus signs in manpages
- 1013: [NumPyExampleDocstring] link was pointing to raw file
- 1011: Add hashed password support.
- 1005: Quick fix for os.system requiring str parameter
- 994: Allow latex formulas in HTML output
- 955: Websocket Adjustments
- 979: use system_raw in terminal, even on Windows
- 989: fix arguments for commands in _process_posix
- 991: Show traceback, continuing to start kernel if pylab init fails
- 981: Split likely multiline text when writing JSON notebooks
- 957: allow change of png DPI in inline backend
- 968: add wantDirectory to ipdoctest, so that directories will be checked for e

- 984: Do not expose variables defined at startup to %who etc.
- 985: Fixes for parallel code on Python 3
- 963: disable calltips in PySide < 1.0.7 to prevent segfault
- 976: Getting started on what's new
- 929: Multiline history
- 964: Default profile
- 961: Disable the pager for the test suite
- 953: Physics extension
- 950: Add directory for startup files
- 940: allow setting HistoryManager.hist_file with config
- 948: Monkeypatch Tornado 2.1.1 so it works with Google Chrome 16.
- 916: Run p (https://github.com/ipython/ipython/pull/901)
- 923: %config magic
- 920: unordered iteration of AsyncMapResults (+ a couple fixes)
- 941: Follow-up to 387dcd6a, *_rl.__doc__* is *None* with pyreadline
- 931: read-only notebook mode
- 921: Show invalid config message on TraitErrors during init
- 815: Fix #481 using custom qt4 input hook
- 936: Start webbrowser in a thread. Prevents lockup with Chrome.
- 937: add dirty trick for readline import on OSX
- 913: Py3 tests2
- 933: Cancel in qt console closeevent should trigger event.ignore()
- 930: read-only notebook mode
- 910: Make import checks more explicit in %whos
- 926: reincarnate -V cmdline option
- 928: BUG: Set context for font size change shortcuts in ConsoleWidget
- 901: - There is a bug when running the profiler in the magic command (prun) with python3
- 912: Add magic for cls on windows. Fix for #181.
- 905: enable %gui/%pylab magics in the Kernel
- 909: Allow IPython to run without sqlite3
- 887: Qtconsole menu
- 895: notebook download implies save

- 896: Execfile
- 899: Brian's Notebook work
- 892: don't close figures every cycle with inline matplotlib backend
- 893: Adding clear_output to kernel and HTML notebook
- 789: Adding clear_output to kernel and HTML notebook.
- 898: Don't pass unicode sys.argv with %run or *ipython script.py*
- 897: Add tooltips to the notebook via 'title' attr.
- 877: partial fix for issue #678
- 838: reenable multiline history for terminals
- 872: The constructor of Client() checks for AssertionError in validate_url to open a file instead of connection to a URL if it fails.
- 884: Notebook usability fixes
- 883: User notification if notebook saving fails
- 889: Add drop_by_id method to shell, to remove variables added by extensions.
- 891: Ability to open the notebook in a browser when it starts
- 813: Create menu bar for qtconsole
- 876: protect IPython from bad custom exception handlers
- 856: Backgroundjobs
- 868: Warn user if MathJax can't be fetched from notebook closes #744
- 878: store_history=False default for run_cell
- 824: History access
- 850: Update codemirror to 2.15 and make the code internally more version-agnostic
- 861: Fix for issue #56
- 819: Adding -m option to %run, similar to -m for python interpreter.
- 855: promote aliases and flags, to ensure they have priority over config files
- 862: BUG: Completion widget position and pager focus.
- 847: Allow connection to kernels by files
- 708: Two-process terminal frontend
- 857: make sdist flags work again (e.g. –manifest-only)
- 835: Add Tab key to list of keys that scroll down the paging widget.
- 859: Fix for issue #800
- 848: Python3 setup.py install failiure

- 845: Tests on Python 3
- 802: DOC: extensions: add documentation for the bundled extensions
- 830: contiguous stdout/stderr in notebook
- 761: Windows: test runner fails if repo path (e.g. home dir) contains spaces
- 801: Py3 notebook
- 809: use CFRunLoop directly in *ipython kernel –pylab osx*
- 841: updated old scipy.org links, other minor doc fixes
- 837: remove all trailling spaces
- 834: Issue https://github.com/ipython/ipython/issues/832 resolution
- 746: ENH: extensions: port autoreload to current API
- 828: fixed permissions (sub-modules should not be executable) + added shebang for run_ipy_in_profiler.py
- 798: pexpect & Python 3
- 804: Magic 'range' crash if greater than len(input_hist)
- 821: update tornado dependency to 2.1
- 807: Faciliate ssh tunnel sharing by announcing ports
- 795: Add cluster-id for multiple cluster instances per profile
- 742: Glut
- 668: Greedy completer
- 776: Reworking qtconsole shortcut, add fullscreen
- 790: TST: add future unicode_literals test (#786)
- 775: redirect_in/redirect_out should be constrained to windows only
- 793: Don't use readline in the ZMQShell
- 743: Pyglet
- 774: basic/initial .mailmap for nice shortlog summaries
- 770: #769 (reopened)
- 784: Parse user code to AST using compiler flags.
- 783: always use StringIO, never cStringIO
- 782: flush stdout/stderr on displayhook call
- 622: Make pylab import all configurable
- 745: Don't assume history requests succeed in qtconsole
- 725: don't assume cursor.selectedText() is a string

- 778: don't override execfile on Python 2
- 663: Python 3 compatilibility work
- 762: qtconsole ipython widget's execute_file fails if filename contains spaces or quotes
- 763: Set context for shortcuts in ConsoleWidget
- 722: PyPy compatibility
- 757: ipython.el is broken in 0.11
- 764: fix "–colors=<color>" option in py-python-command-args.
- 758: use ROUTER/DEALER socket names instead of XREP/XREQ
- 736: enh: added authentication ability for webapp
- 748: Check for tornado before running frontend.html tests.
- 754: restore msg_id/msg_type aliases in top level of msg dict
- 769: Don't treat bytes objects as json-safe
- 753: DOC: msg['msg_type'] removed
- 766: fix "–colors=<color>" option in py-python-command-args.
- 765: fix "–colors=<color>" option in py-python-command-args.
- 741: Run PyOs_InputHook in pager to keep plot windows interactive.
- 664: Remove ipythonrc references from documentation
- 750: Tiny doc fixes
- 433: ZMQ terminal frontend
- 734: Allow %magic argument filenames with spaces to be specified with quotes under win32
- 731: respect encoding of display data from urls
- 730: doc improvements for running notebook via secure protocol
- 729: use null char to start markdown cell placeholder
- 727: Minor fixes to the htmlnotebook
- 726: use bundled argparse if system argparse is < 1.1
- 705: Htmlnotebook
- 723: Add 'import time' to IPython/parallel/apps/launcher.py as time.sleep is called without time being imported
- 714: Install mathjax for offline use
- 718: Underline keyboard shortcut characters on appropriate buttons
- 717: Add source highlighting to markdown snippets
- 716: update EvalFormatter to allow arbitrary expressions

- 712: Reset execution counter after cache is cleared
- 713: Align colons in html notebook help dialog
- 709: Allow usage of '.' in notebook names
- 706: Implement static publishing of HTML notebook
- 674: use argparse to parse aliases & flags
- 679: HistoryManager.get_session_info()
- 696: Fix columnize bug, where tab completion with very long filenames would crash Qt console
- 686: add ssh tunnel support to qtconsole
- 685: Add SSH tunneling to engines
- 384: Allow pickling objects defined interactively.
- 647: My fix rpmlint
- 587: don't special case for py3k+numpy
- 703: make config-loading debug messages more explicit
- 699: make calltips configurable in qtconsole
- 666: parallel tests & extra readline escapes
- 683: BF - allow nose with-doctest setting in environment
- 689: Protect ipkernel from bad messages
- 702: Prevent ipython.py launcher from being imported.
- 701: Prevent ipython.py from being imported by accident
- 670: check for writable dirs, not just existence, in utils.path
- 579: Sessionwork
- 687: add *ipython kernel* for starting just a kernel
- 627: Qt Console history search
- 646: Generate package list automatically in find_packages
- 660: i658
- 659: don't crash on bad config files

Regular issues (258):

- 1177: UnicodeDecodeError in py3compat from "xlrd??"
- 1094: Tooltip doesn't show constructor docstrings
- 1170: double pylab greeting with c.InteractiveShellApp.pylab = "tk" in zmqconsole
- 1166: E-mail cpaste broken
- 1164: IPython qtconsole (0.12) can't export to html with external png

- 1103: %loadpy should cut out encoding declaration
- 1156: Notebooks downloaded as Python files require a header stating the encoding
- 1157: Ctrl-C not working when GUI/pylab integration is active
- 1154: We should be less aggressive in de-registering post-execution functions
- 1134: "select-all, kill" leaves qtconsole in unusable state
- 1148: A lot of testerrors
- 803: Make doctests work with Python 3
- 1119: Start menu shortcuts not created in Python 3
- 1136: The embedding machinery ignores user_ns
- 607: Use the new IPython logo/font in the notebook header
- 755: qtconsole ipython widget's execute_file fails if filename contains spaces or quotes
- 1115: shlex_split should return unicode
- 1109: timeit with string ending in space gives "ValueError: No closing quotation"
- 1142: Install problems
- 700: Some SVG images render incorrectly in htmlnotebook
- 1117: quit() doesn't work in terminal
- 1111: ls broken after merge of #1089
- 1104: Prompt spacing weird
- 1124: Seg Fault 11 when calling PySide using "run" command
- 1088: QtConsole : can't copy from pager
- 568: Test error and failure in IPython.core on windows
- 1112: testfailure in IPython.frontend on windows
- 1102: magic in IPythonDemo fails when not located at top of demo file
- 629: r and b in qtconsole don't behave as expected
- 1080: Notebook: tab completion should close on "("
- 973: Qt Console close dialog and on-top Qt Console
- 1087: QtConsole xhtml/Svg export broken ?
- 1067: Parallel test suite hangs on Python 3
- 1018: Local mathjax breaks install
- 993: *raw_input* redirection to foreign kernels is extremely brittle
- 1100: ipython3 traceback unicode issue from extensions
- 1071: Large html-notebooks hang on load on a slow machine

- 89: %pdoc np.ma.compress shows docstring twice
- 22: Include improvements from anythingipython.el
- 633: Execution count & SyntaxError
- 1095: Uncaught TypeError: Object has no method 'remove_and_cancell_tooltip'
- 1075: We're ignoring prompt customizations
- 1086: Can't open qtconsole from outside source tree
- 1076: namespace changes broke *foo.*bar*?* syntax
- 1074: pprinting old-style class objects fails (TypeError: 'tuple' object is not callable)
- 1063: IPython.utils test error due to missing unicodedata module
- 592: Bug in argument parsing for %run
- 378: Windows path escape issues
- 1068: Notebook tab completion broken in Firefox
- 75: No tab completion after "/
- 103: customizable cpaste
- 324: Remove code in IPython.testing that is not being used
- 131: Global variables not seen by cprofile.run()
- 851: IPython shell swallows exceptions in certain circumstances
- 882: ipython freezes at start if IPYTHONDIR is on an NFS mount
- 1057: Blocker: Qt console broken after "all magics" menu became dynamic
- 1027: ipython does not like white space at end of file
- 1058: New bug: Notebook asks for confirmation to leave even saved pages.
- 1061: rep (magic recall) under pypy
- 1047: Document the notebook format
- 102: Properties accessed twice for classes defined interactively
- 16: %store raises exception when storing compiled regex
- 67: tab expansion should only take one directory level at the time
- 62: Global variables undefined in interactive use of embedded ipython shell
- 57: debugging with ipython does not work well outside ipython
- 38: Line entry edge case error
- 980: Update parallel docs for new parallel architecture
- 1017: Add small example about ipcluster/ssh startup
- 1041: Proxy Issues

- 967: KernelManagers don't use zmq eventloop properly
- 1055: "All Magics" display on Ubuntu
- 1054: ipython explodes on syntax error
- 1051: ipython3 set_next_input() failure
- 693: "run -i" no longer works after %reset in terminal
- 29: cPickle works in standard interpreter, but not in IPython
- 1050: ipython3 broken by commit 8bb887c8c2c447bf7
- 1048: Update docs on notebook password
- 1046: Searies of questions/issues?
- 1045: crash when exiting - previously launched embedded sub-shell
- 1043: pylab doesn't work in qtconsole
- 1044: run -p doesn't work in python 3
- 1010: emacs freezes when ipython-complete is called
- 82: Update devel docs with discussion about good changelogs
- 116: Update release management scipts and release.revision for git
- 1022: Pylab banner shows up with first cell to execute
- 787: Keyboard selection of multiple lines in the notebook behaves inconsistently
- 1037: notepad + jsonlib: TypeError: Only whitespace may be used for indentation.
- 970: Default home not writable, %HOME% does not help (windows)
- 747: HOMESHARE not a good choice for "writable homedir" on Windows
- 810: cleanup utils.path.get_home_dir
- 2: Fix the copyright statement in source code files to be accurate
- 1031: <esc> on Firefox crash websocket
- 684: %Store eliminated in configuration and magic commands in 0.11
- 1026: BUG: wrong default parameter in ask_yes_no
- 880: Better error message if %paste fails
- 1024: autopx magic broken
- 822: Unicode bug in Itpl when expanding shell variables in syscalls with !
- 1009: Windows: regression in cd magic handling of paths
- 833: Crash python with matplotlib and unequal length arrays
- 695: Crash handler initialization is too aggressive
- 1000: Remove duplicates when refilling readline history

- 992: Interrupting certain matplotlib operations leaves the inline backend 'wedged'
- 942: number traits should cast if value doesn't change
- 1006: ls crashes when run on a UNC path or with non-ascii args
- 944: Decide the default image format for inline figures: SVG or PNG?
- 842: Python 3 on Windows (pyreadline) - expected an object with the buffer interface
- 1002: ImportError due to incorrect version checking
- 1001: Ipython "source" command?
- 954: IPython embed doesn't respect namespaces
- 681: pdb freezes inside qtconsole
- 698: crash report "TypeError: can only concatenate list (not "unicode") to list"
- 978: ipython 0.11 buffers external command output till the cmd is done
- 952: Need user-facing warning in the browser if websocket connection fails
- 988: Error using idlsave
- 990: ipython notebook - kernel dies if matplotlib is not installed
- 752: Matplotlib figures showed only once in notebook
- 54: Exception hook should be optional for embedding IPython in GUIs
- 918: IPython.frontend tests fail without tornado
- 986: Views created with c.direct_view() fail
- 697: Filter out from %who names loaded at initialization time
- 932: IPython 0.11 quickref card has superfluous "%recall and"
- 982: png files with executable permissions
- 914: Simpler system for running code after InteractiveShell is initialised
- 911: ipython crashes on startup if readline is missing
- 971: bookmarks created in 0.11 are corrupt in 0.12
- 974: object feature tab-completion crash
- 939: ZMQShell always uses default profile
- 946: Multi-tab Close action should offer option to leave all kernels alone
- 949: Test suite must not require any manual interaction
- 643: enable gui eventloop integration in ipkernel
- 965: ipython is crashed without launch.(python3.2)
- 958: Can't use os X clipboard on with qtconsole
- 962: Don't require tornado in the tests

- 960: crash on syntax error on Windows XP
- 934: The latest ipython branch doesn't work in Chrome
- 870: zmq version detection
- 943: HISTIGNORE for IPython
- 947: qtconsole segfaults at startup
- 903: Expose a magic to control config of the inline pylab backend
- 908: bad user config shouldn't crash IPython
- 935: Typing *break* causes IPython to crash.
- 869: Tab completion of ~/ shows no output post 0.10.x
- 904: whos under pypy1.6
- 773: check_security_dir() and check_pid_dir() fail on network filesystem
- 915: OS X Lion Terminal.app line wrap problem
- 886: Notebook kernel crash when specifying –notebook-dir on commandline
- 636: debugger.py: pydb broken
- 808: Ctrl+C during %reset confirm message crash Qtconsole
- 927: Using return outside a function crashes ipython
- 919: Pop-up segfault when moving cursor out of qtconsole window
- 181: cls command does not work on windows
- 917: documentation typos
- 818: %run does not work with non-ascii characeters in path
- 907: Errors in custom completer functions can crash IPython
- 867: doc: notebook password authentication howto
- 211: paste command not working
- 900: Tab key should insert 4 spaces in qt console
- 513: [Qt console] cannot insert new lines into console functions using tab
- 906: qtconsoleapp 'parse_command_line' doen't like –existing anymore
- 638: Qt console –pylab=inline and getfigs(), etc.
- 710: unwanted unicode passed to args
- 436: Users should see tooltips for all buttons in the notebook UI
- 207: ipython crashes if atexit handler raises exception
- 692: use of Tracer() when debugging works but gives error messages
- 690: debugger does not print error message by default in 0.11

- 571: history of multiline entries
- 749: IPython.parallel test failure under Windows 7 and XP
- 890: ipclusterapp.py - helep
- 885: *ws-hostname* alias not recognized by notebook
- 881: Missing manual.pdf?
- 744: cannot create notebook in offline mode if mathjax not installed
- 865: Make tracebacks from %paste show the code
- 535: exception unicode handling in %run is faulty in qtconsole
- 817: iPython crashed
- 799: %edit magic not working on windows xp in qtconsole
- 732: QTConsole wrongly promotes the index of the input line on which user presses Enter
- 662: ipython test failures on Mac OS X Lion
- 650: Handle bad config files better
- 829: We should not insert new lines after all print statements in the notebook
- 874: ipython-qtconsole: pyzmq Version Comparison
- 640: matplotlib macosx windows don't respond in qtconsole
- 624: ipython intermittently segfaults when figure is closed (Mac OS X)
- 871: Notebook crashes if a profile is used
- 56: Have %cpaste accept also Ctrl-D as a termination marker
- 849: Command line options to not override profile options
- 806: Provide single-port connection to kernels
- 691: [wishlist] Automatically find existing kernel
- 688: local security vulnerability: all ports visible to any local user.
- 866: DistributionNotFound on running ipython 0.11 on Windows XP x86
- 673: raw_input appears to be round-robin for qtconsole
- 863: Graceful degradation when home directory not writable
- 800: Timing scripts with run -t -N <N> fails on report output
- 858: Typing 'continue' makes ipython0.11 crash
- 840: all processes run on one CPU core
- 843: "import braces" crashes ipython
- 836: Strange Output after IPython Install
- 839: Qtconsole segfaults when mouse exits window with active tooltip

- 827: Add support for checking several limits before running task on engine
- 826: Add support for creation of parallel task when no engine is running
- 832: Improve error message for %logstop
- 831: %logstart in read-only directory forbid any further command
- 814: ipython does not start – DistributionNotFound
- 794: Allow >1 controller per profile
- 820: Tab Completion feature
- 812: Qt console crashes on Ubuntu 11.10
- 816: Import error using Python 2.7 and dateutil2.0 No module named _thread
- 756: qtconsole Windows fails to print error message for '%run nonexistent_file'
- 651: Completion doesn't work on element of a list
- 617: [qtconsole] %hist doesn't show anything in qtconsole
- 786: from __future__ import unicode_literals does not work
- 779: Using irunner from virtual evn uses systemwide ipython
- 768: codepage handling of output from scripts and shellcommands are not handled properly by qt-console
- 785: Don't strip leading whitespace in repr() in notebook
- 737: in pickleshare.py line52 should be "if not os.path.isdir(self.root):"?
- 738: in ipthon_win_post_install.py line 38
- 777: print(…, sep=…) raises SyntaxError
- 728: ipcontroller crash with MPI
- 780: qtconsole Out value prints before the print statements that precede it
- 632: IPython Crash Report (0.10.2)
- 253: Unable to install ipython on windows
- 80: Split IPClusterApp into multiple Application subclasses for each subcommand
- 34: non-blocking pendingResult partial results
- 739: Tests fail if tornado not installed
- 719: Better support Pypy
- 667: qtconsole problem with default pylab profile
- 661: ipythonrc referenced in magic command in 0.11
- 665: Source introspection with ?? is broken
- 724: crash - ipython qtconsole, %quickref

- 655: ipython with qtconsole crashes
- 593: HTML Notebook Prompt can be deleted . . .
- 563: use argparse instead of kvloader for flags&aliases
- 751: Tornado version greater than 2.0 needed for firefox 6
- 720: Crash report when importing easter egg
- 740: Ctrl-Enter clears line in notebook
- 772: ipengine fails on Windows with "XXX lineno: 355, opcode: 0"
- 771: Add python 3 tag to setup.py
- 767: non-ascii in __doc__ string crashes qtconsole kernel when showing tooltip
- 733: In Windows, %run fails to strip quotes from filename
- 721: no completion in emacs by ipython(ipython.el)
- 669: Do not accept an ipython_dir that's not writeable
- 711: segfault on mac os x
- 500: "RuntimeError: Cannot change input buffer during execution" in console_widget.py
- 707: Copy and paste keyboard shortcuts do not work in Qt Console on OS X
- 478: PyZMQ's use of memoryviews breaks reconstruction of numpy arrays
- 694: Turning off callout tips in qtconsole
- 704: return kills IPython
- 442: Users should have intelligent autoindenting in the notebook
- 615: Wireframe and implement a project dashboard page
- 614: Wireframe and implement a notebook dashboard page
- 606: Users should be able to use the notebook to import/export a notebook to .py or .rst
- 604: A user should be able to leave a kernel running in the notebook and reconnect
- 298: Users should be able to save a notebook and then later reload it
- 649: ipython qtconsole (v0.11): setting "c.IPythonWidget.in_prompt = '>>> ' crashes
- 672: What happened to Exit?
- 658: Put the InteractiveShellApp section first in the auto-generated config files
- 656: [suggestion] dependency checking for pyqt for Windows installer
- 654: broken documentation link on download page
- 653: Test failures in IPython.parallel

2.8 0.11 Series

2.8.1 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.

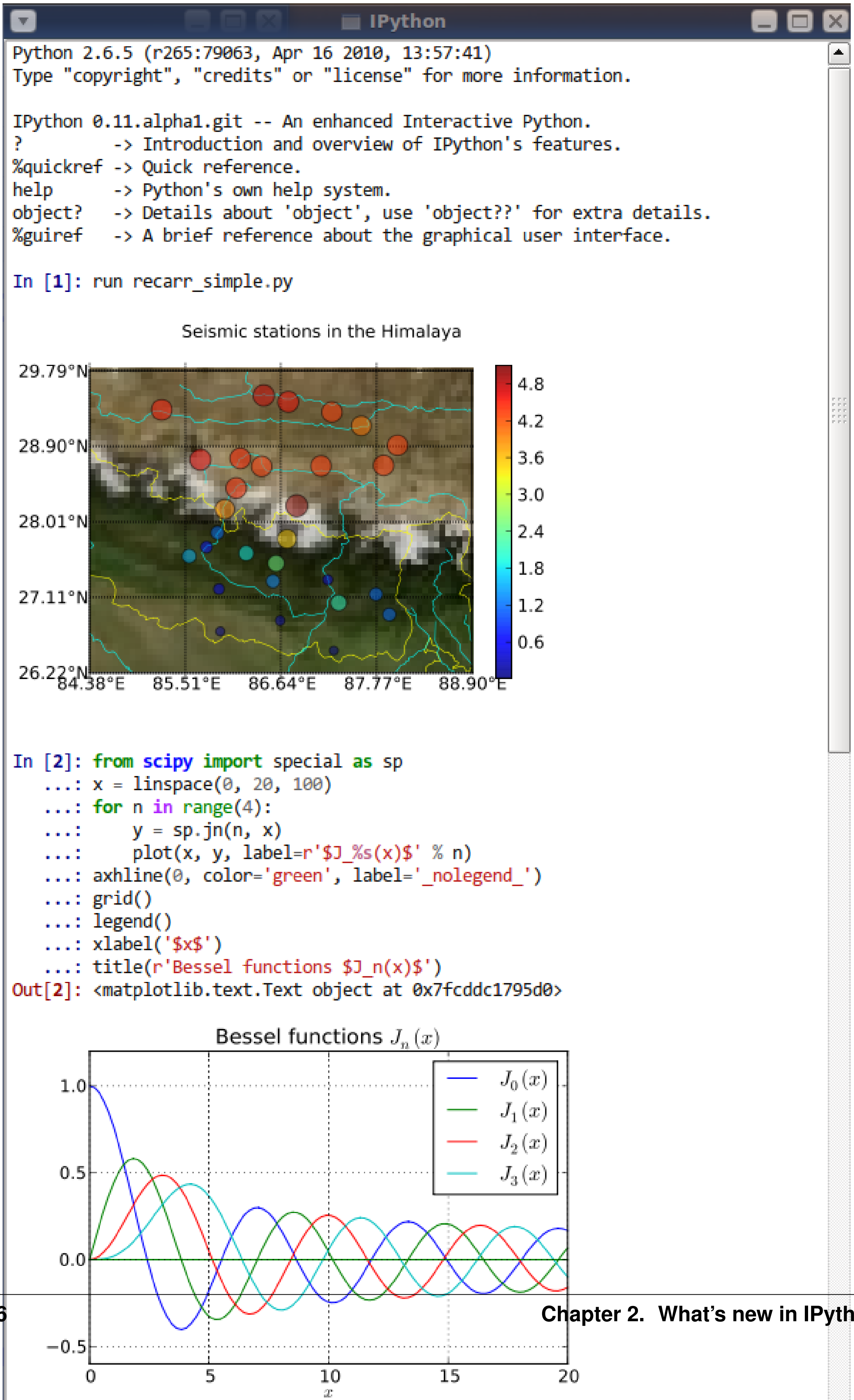
Note: This undercounts the number of issues closed in this development cycle, since we only moved to github for issue tracking in May 2010, but we have no way of collecting statistics on the number of issues closed in the old Launchpad bug tracker prior to that.

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.



```
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.
%guiref   -> 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>
```

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.

Refactoring

As of this release, a signifiant 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 `ipythyon --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"€ø"`) now works properly on all platforms. However, entering these in byte/string literals (`"€ø"`) 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.

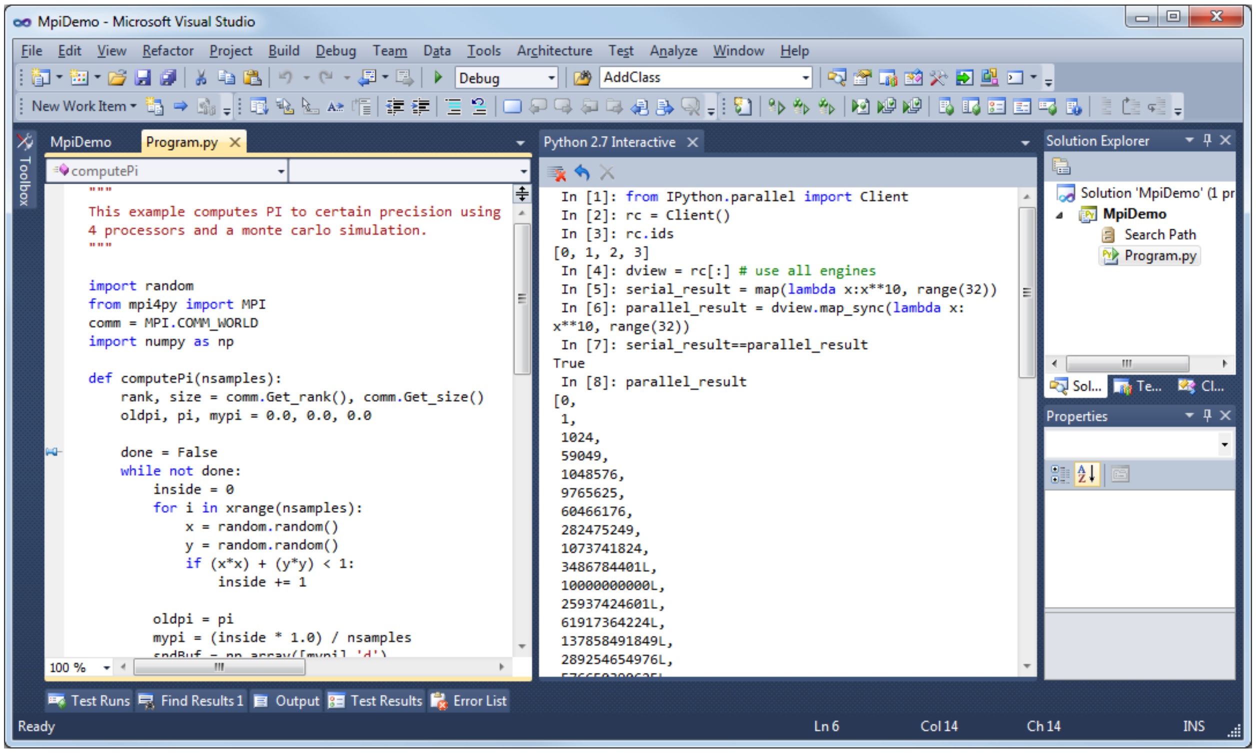
Integration with Microsoft Visual Studio

IPython can be used as the interactive shell in the Python plugin for Microsoft Visual Studio, as seen here:

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 a the logger of the currently running Application with:



```
"""
This example computes PI to certain precision using
4 processors and a monte carlo simulation.
"""

import random
from mpi4py import MPI
comm = MPI.COMM_WORLD
import numpy as np

def computePi(nsamples):
    rank, size = comm.Get_rank(), comm.Get_size()
    oldpi, pi, mypi = 0.0, 0.0, 0.0

    done = False
    while not done:
        inside = 0
        for i in xrange(nsamples):
            x = random.random()
            y = random.random()
            if (x*x) + (y*y) < 1:
                inside += 1

        oldpi = pi
        mypi = (inside * 1.0) / nsamples
```

```
 In [1]: from IPython.parallel import Client
 In [2]: rc = Client()
 In [3]: rc.ids
[0, 1, 2, 3]
 In [4]: dview = rc[:] # use all engines
 In [5]: serial_result = map(lambda x:x**10, range(32))
 In [6]: parallel_result = dview.map_sync(lambda x:
x**10, range(32))
 In [7]: serial_result==parallel_result
True
 In [8]: parallel_result
[0,
 1,
 1024,
 59049,
 1048576,
 9765625,
 60466176,
 282475249,
 1073741824,
 3486784401L,
 10000000000L,
 25937424601L,
 61917364224L,
 137858491849L,
 289254654976L,
```

Fig. 2.6: IPython console embedded in Microsoft Visual Studio.

```
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 `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 `__builtin__`.
- `Struct.__init__()` no longer takes *None* as its first argument. It must be a `dict` or `Struct`.
- `ipmagic()` has been renamed `()`
- The functions `ipmagic()` and `ipalias()` have been removed from `__builtins__`.
- The references to the global `InteractiveShell` instance (`_ip`, and `__IP`) have been removed from the user's namespace. They are replaced by a new function called `get_ipython()` that returns the current `InteractiveShell` instance. This function is injected into the user's namespace and is now the main way of accessing 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:

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

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:

```
astyle.py                 ipy_defaults.py          ipy_vimserver.py
dtutils.py                ipy_kitcfg.py            numeric_formats.py
Gnuplot2.py               ipy_legacy.py            numutils.py
GnuplotInteractive.py     ipy_p4.py                outputtrap.py
GnuplotRuntime.py         ipy_profile_none.py      PhysicalQInput.py
ibrowse.py                ipy_profile_numpy.py     PhysicalQInteractive.py
igrid.py                  ipy_profile_scipy.py     quitter.py*
ipipe.py                  ipy_profile_sh.py        scitedirector.py
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 codebase 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 <laurent.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.

2.9 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: Getcwdu
- 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 AssertionError
- 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.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 varibles 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.ipy 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 evaulation 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: multline block scrolling
- 46: Input to %timeit is not preparsed
- 285: ipcluster local -n 4 fails
- 205: In the Qt console, Tab should insert 4 spaces when not completing
- 145: Bug on MSW sytems: 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 referrences
- 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 "_ip.magic"
- 135: crash on invalid print
- 69: Usage of "mycluster" profile in docs and examples
- 37: Fix colors in output of ResultList on Windows

2.10 0.10 series

2.10.1 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 ommitted your name and we'll gladly fix this in the notes for the future):

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

2.10.2 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 reSructuredText documents as containing IPython syntax (including figures) and the will be executed during the build:

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

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

- Various fixes to the standalone ipython-wx application.

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

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

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

2.10.3 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)

2.11 0.9 series

2.11.1 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.

2.11.2 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 arhictecture of the various IPython components. We will continue to develop this over the next few releases as a model for GUI components that use IPython.
- Another GUI frontend, Cocoa based (Cocoa is the OSX native GUI framework), authored by Barry Wark. Currently the WX and the Cocoa ones have slightly different internal organizations, but the whole team is working on finding what the right abstraction points are for a unified codebase.
- As part of the frontend work, Barry Wark also implemented an experimental event notification system that various ipython components can use. In the next release the implications and use patterns of this system regarding the various GUI options will be worked out.
- IPython finally has a full test system, that can test docstrings with IPython-specific functionality. There are still a few pieces missing for it to be widely accessible to all users (so they can run the test suite at any time and report problems), but it now works for the developers. We are working hard on continuing to improve it, as this was probably IPython's major Achilles heel (the lack of proper test coverage made it effectively impossible to do large-scale refactoring). The full test suite can now be run using the **`iptest`** command line program.
- The notion of a task has been completely reworked. An *ITask* interface has been created. This interface defines the methods that tasks need to implement. These methods are now responsible for things like submitting tasks and processing results. There are two basic task types: `IPython.kernel.task.StringTask` (this is the old *Task* object, but renamed) and the new `IPython.kernel.task.MapTask`, which is based on a function.
- A new interface, `IPython.kernel.mapper.IMapper` has been defined to standardize the idea of a *map* method. This interface has a single *map* method that has the same syntax as the built-in *map*. We have also defined a *mapper* factory interface that creates objects that implement `IPython.kernel.mapper.IMapper` for different controllers. Both the multiengine and task controller now have mapping capabilties.
- The parallel function capabilities have been reworks. The major changes are that i) there is now an *@parallel* magic that creates parallel functions, ii) the syntax for multiple variable follows that of *map*, iii) both the multiengine and task controller now have a parallel function implementation.
- All of the parallel computing capabilities from *ipython1-dev* have been merged into IPython proper. This resulted in the following new subpackages: `IPython.kernel`, `IPython.kernel.core`, `IPython.config`, `IPython.tools` and `IPython.testing`.
- As part of merging in the *ipython1-dev* stuff, the *setup.py* script and friends have been completely refactored. Now we are checking for dependencies using the approach that matplotlib uses.
- The documentation has been completely reorganized to accept the documentation from *ipython1-dev*.
- We have switched to using Foolscap for all of our network protocols in `IPython.kernel`. This gives us secure connections that are both encrypted and authenticated.

- We have a brand new *COPYING.txt* files that describes the IPython license and copyright. The biggest change is that we are putting "The IPython Development Team" as the copyright holder. We give more details about exactly what this means in this file. All developer should read this and use the new banner in all IPython source code files.
- sh profile: ./foo runs foo as system command, no need to do !./foo anymore
- String lists now support `sort(field, nums = True)` method (to easily sort system command output). Try it with `a = !ls -l ; a.sort(1, nums=1)`.
- '%cpaste foo' now assigns the pasted block as string list, instead of string
- The ipcluster script now run by default with no security. This is done because the main usage of the script is for starting things on localhost. Eventually when ipcluster is able to start things on other hosts, we will put security back.
- 'cd –foo' searches directory history for string foo, and jumps to that dir. Last part of dir name is checked first. If no matches for that are found, look at the whole path.

Bug fixes

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

Backwards incompatible changes

- The `clusterfile` options of the **`ipcluster`** command has been removed as it was not working and it will be replaced soon by something much more robust.
- The `IPython.kernel` configuration now properly find the user's IPython directory.
- In ipapi, the `make_user_ns()` function has been replaced with `make_user_namespaces()`, to support dict subclasses in namespace creation.
- `IPython.kernel.client.Task` has been renamed `IPython.kernel.client.StringTask` to make way for new task types.
- The keyword argument *style* has been renamed *dist* in *scatter*, *gather* and *map*.
- Renamed the values that the rename *dist* keyword argument can have from *'basic'* to *'b'*.

- IPython has a larger set of dependencies if you want all of its capabilities. See the *setup.py* script for details.
- The constructors for `IPython.kernel.client.MultiEngineClient` and `IPython.kernel.client.TaskClient` no longer take the (ip,port) tuple. Instead they take the filename of a file that contains the FURL for that client. If the FURL file is in your IPYTHONDIR, it will be found automatically and the constructor can be left empty.
- The asynchronous clients in `IPython.kernel.asyncclient` are now created using the factory functions `get_multiengine_client()` and `get_task_client()`. These return a *Deferred* to the actual client.
- The command line options to *ipcontroller* and *ipengine* have changed to reflect the new 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 not 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 docuementation. 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`.

2.12 0.8 series

2.12.1 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.

2.12.2 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.

2.12.3 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.

2.12.4 Older releases

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

CHAPTER 3

Installation

IPython requires Python 2.6, 2.7, or 3.2.

Note: If you need to use Python 2.5, you can find an old version (0.10) of IPython here.

3.1 Quickstart

If you have `setuptools`, the quickest way to get up and running with IPython is:

```
$ easy_install ipython[all]
```

This will download and install IPython and its main optional dependencies:

- jinja2, needed for the notebook
- sphinx, needed for nbconvert
- pyzmq, needed for IPython's parallel computing features, qt console and notebook
- pygments, used by nbconvert and the Qt console for syntax highlighting
- tornado, needed by the web-based notebook
- nose, used by the test suite
- readline (on OS X) or pyreadline (on Windows), needed for the terminal

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

```
$ iptest
```

Note:

```
$ pip install ipython[all]
```

will also work in many cases, but it will ignore the binary eggs of packages such as pyzmq and readline, which may be required for some users on Windows or OS X.

3.2 Overview

This document describes in detail the steps required to install IPython, and its various optional dependencies. For a few quick ways to get started with package managers or full Python distributions, see the install page of the IPython website.

IPython is organized into a number of subpackages, 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.

IPython and most dependencies can be installed via **`easy_install`**, provided by the `setuptools` package, or **`pip`**. In many scenarios, this is the most simplest method of installing Python packages. More information about `setuptools` can be found on its PyPI page.

Note: On Windows, IPython *requires* `setuptools`. We hope to change this in the future, but for now on Windows, you *must* install `setuptools` to use IPython.

More general information about installing Python packages can be found in Python's documentation.

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 or pip

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

```
$ pip install ipython
```

That's it.

Note: Many prefer **`pip`** to **`easy_install`**, but it ignores eggs (binary Python packages). This mainly affects pyzmq and readline, which are compiled packages and provide binary eggs. If you use **`pip`** to install these packages, it will always compile from source, which may not succeed.

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

As mentioned above, on Windows, IPython requires `setuptools`, and it also requires the PyReadline library to properly support coloring and keyboard management (features that the default windows console doesn't have). So on Windows, the installation procedure is:

1. Install setuptools.
2. Install pyreadline. You can use the command `easy_install pyreadline` from a terminal, or the binary installer appropriate for your platform from the PyPI page.
3. Install IPython itself, which you can download from PyPI or from our site. Note that on Windows 7, you *must* right-click and 'Run as administrator' for the Start menu shortcuts to be created.

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.

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

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

As indicated above, on Windows, PyReadline is a *mandatory* dependency. 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.

On OSX, if you are using the built-in Python shipped by Apple, you will be missing a full readline implementation as Apple ships instead a library called `libedit` that provides only some of readline's functionality. 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.

When IPython is installed with `setuptools`, (e.g. using the `easy_install` command), 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.

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`** or **`pip`**:

```
$ pip install nose
```

Another way of getting this is to do:

```
$ pip 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. 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:

```
$ pip install pexpect
```

Note: On Python 3, you should actually install `pexpect-u`, a unicode-safe fork of pexpect.

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

3.5 Dependencies for IPython.parallel (parallel computing)

IPython.parallel provides a nice architecture for parallel computing, with a focus on fluid interactive workflows. 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.

3.6 Dependencies for IPython.kernel.zmq

3.6.1 pyzmq

IPython 0.11 introduced some new functionality, including a two-process execution model using ZeroMQ for communication. 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.kernel.zmq depends on pyzmq >= 2.1.4.

3.7 Dependencies for the IPython QT console

3.7.1 pyzmq

Like the `IPython.parallel` package, the QT Console requires ZeroMQ and PyZMQ.

3.7.2 Qt

Also with 0.11, a new GUI was added using the work in `IPython.kernel.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.

3.7.3 pygments

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

3.8 Dependencies for the IPython HTML notebook

The IPython notebook is a notebook-style web interface to IPython and can be started with the command `ipython notebook`.

3.8.1 pyzmq

Like the `IPython.parallel` and `IPython.frontend.qt.console` packages, the HTML notebook requires ZeroMQ and PyZMQ.

3.8.2 Tornado

The IPython notebook uses the Tornado project for its HTTP server. Tornado 2.1 is required, in order to support current versions of browsers, due to an update to the websocket protocol.

3.8.3 Jinja

The IPython notebook uses the Jinja templating tool to render HTML pages.

3.8.4 MathJax

The IPython notebook uses the MathJax Javascript library for rendering LaTeX in web browsers. Because MathJax is large, we don't include it with IPython. Normally IPython will load MathJax from a CDN, but if you have a slow network connection, or want to use LaTeX without an internet connection at all, you can install MathJax locally.

A quick and easy method is to install it from a python session:

```python
from IPython.external.mathjax import install_mathjax
install_mathjax()
```

If you need tighter configuration control, you can download your own copy of MathJax from http://www.mathjax.org/download/ - use the MathJax-2.0 link. When you have the file stored locally, install it with:

```
python -m IPython.external.mathjax /path/to/source/mathjax-MathJax-v2.0-20-g07669ac.zip
```

For unusual needs, IPython can tell you what directory it wants to find MathJax in:

```
python -m IPython.external.mathjax -d /some/other/mathjax
```

By default Mathjax will be installed in your ipython profile directory, but you can make system wide install, please refer to the documentation and helper function of `IPython.external.mathjax`

3.8.5 Browser Compatibility

The IPython notebook is officially supported on the following browers:

- Chrome 13
- Safari 5
- Firefox 6

The is mainly due to the notebook's usage of WebSockets and the flexible box model.

The following browsers are unsupported:

- Safari < 5
- Firefox < 6
- Chrome < 13
- Opera (any): CSS issues, but execution might work
- Internet Explorer < 10

The following specific combinations are known **NOT** to work:

- Safari, IPython 0.12, tornado 2.2.0
- Safari with HTTPS connection to notebook and an untrusted certificate (websockets will fail)
- The [diigo Chrome extension](http://help.diigo.com/tools/chrome-extension) seems to interfere with scrolling

There are some early reports that the Notebook works on Internet Explorer 10, but we expect there will be some CSS issues related to the flexible box model.

3.9 Dependencies for nbconvert (converting notebooks to various formats)

3.9.1 pandoc

The most important dependency of nbconvert is Pandoc, a document format translation program. This is not a Python package, so it cannot be expressed as a regular IPython dependency with setuptools.

To install pandoc on Linux, you can generally use your package manager:

```
sudo apt-get install pandoc
```

On other platforms, you can get pandoc from their website.

CHAPTER 4

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 The four most helpful commands

The four most helpful commands, as well as their brief description, is shown to you in a banner, every time you start IPython:

command	description
?	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.

4.1.2 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.

4.1.3 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`

4.1.4 Magic functions

IPython has a set of predefined 'magic functions' that you can call with a command line style syntax. There are two kinds of magics, line-oriented and cell-oriented. **Line magics** are prefixed with the `%` character and work much like OS command-line calls: they get as an argument the rest of the line, where arguments are passed without parentheses or quotes. **Cell magics** are prefixed with a double `%%`, and they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument.

The following examples show how to call the builtin `timeit` magic, both in line and cell mode:

```
In [1]: %timeit range(1000)
100000 loops, best of 3: 7.76 us per loop

In [2]: %%timeit x = range(10000)
   ...: max(x)
   ...:
1000 loops, best of 3: 223 us per loop
```

The builtin magics include:

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

You can always call them using the `%` prefix, and if you're calling a line magic on a line by itself, you can omit even that:

```
run thescript.py
```

You can toggle this behavior by running the `%automagic` magic. Cell magics must always have the `%%` prefix.

A more detailed explanation of the magic system can be obtained by calling `%magic`, and for more details on any magic function, call `%somemagic?` to read its docstring. To see all the available magic functions, call `%lsmagic`.

See also:

Cell magics example notebook

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. This 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`.

4.1.5 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`, keyed by the prompt numbers, 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.

4.1.6 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.

4.1.7 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*.

Startup Files

If you want some code to be run at the beginning of every IPython session, the easiest way is to add Python (.py) or IPython (.ipy) scripts to your `profile_default/startup/` directory. Files here will be executed as soon as the IPython shell is constructed, before any other code or scripts you have specified. The files will be run in order of their names, so you can control the ordering with prefixes, like `10-myimports.py`.

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. It also keeps the object out of the output cache, so if you're working with large temporary objects, they'll be released from memory sooner.

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-zD.py

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

hello - this is a temporary file

IPython version control at work :)

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

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

4.3 IPython reference

4.3.1 Command-line usage

You start IPython with the command:

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

Note: For IPython on Python 3, use `ipython3` in place of `ipython`.

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 configuration files for details on those. There are separate configuration files for each profile, and the files look like "ipython_config.py" or "ipython_config_<frontendname>.py". Profile directories look like "profile_profilename" and are typically installed in the IPYTHONDIR 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
```

Command-line Options

To see the options IPython accepts, use `ipython --help` (and you probably should run the output through a pager such as `ipython --help | less` for more convenient reading). This shows all the options that have a single-word alias to control them, but IPython lets you configure all of its objects from the command-line by passing the full class name and a corresponding value; type `ipython --help-all` to see this full list. For example:

```
ipython --matplotlib qt
```

is equivalent to:

```
ipython --TerminalIPythonApp.matplotlib='qt'
```

Note that in the second form, you *must* use the equal sign, as the expression is evaluated as an actual Python assignment. While in the above example the short form is more convenient, only the most common options have a short form, while any configurable variable in IPython can be set at the command-line by using the long form. This long form is the same syntax used in the configuration files, if you want to set these options permanently.

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.

Lines that begin with `%%` signal a *cell magic*: they take as arguments not only the rest of the current line, but all lines below them as well, in the current execution block. Cell magics can in fact make arbitrary modifications to the input they receive, which need not even be valid Python code at all. They receive the whole block as a single string.

As a line magic example, the `%cd` magic works just like the OS command of the same name:

```
In [8]: %cd
/home/fperez
```

The following uses the builtin `timeit` in cell mode:

```
In [10]: %%timeit x = range(10000)
    ...: min(x)
    ...: max(x)
    ...:
1000 loops, best of 3: 438 us per loop
```

In this case, `x = range(10000)` is called as the line argument, and the block with `min(x)` and `max(x)` is called as the cell body. The `timeit` magic receives both.

If you have 'automagic' enabled (as it by default), you don't need to type in the single `%` explicitly for line magics; 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':

```
In [9]: cd mydir
/home/fperez/mydir
```

Note that cell magics *always* require an explicit `%%` prefix, automagic calling only works for line magics.

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 "<ipython-input-3-9fedb3aff56c>", 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, automagic works again

In [6]: cd ipython

/home/fperez/ipython
```

Defining your own magics

There are two main ways to define your own magic functions: from standalone functions and by inheriting from a base class provided by IPython: `IPython.core.magic.Magics`. Below we show code you can place in a file that you load from your configuration, such as any file in the `startup` subdirectory of your default IPython profile.

First, let us see the simplest case. The following shows how to create a line magic, a cell one and one that works in both modes, using just plain functions:

```
from IPython.core.magic import (register_line_magic, register_cell_magic,
                                register_line_cell_magic)

@register_line_magic
```

```
def lmagic(line):
    "my line magic"
    return line

@register_cell_magic
def cmagic(line, cell):
    "my cell magic"
    return line, cell

@register_line_cell_magic
def lcmagic(line, cell=None):
    "Magic that works both as %lcmagic and as %%lcmagic"
    if cell is None:
        print "Called as line magic"
        return line
    else:
        print "Called as cell magic"
        return line, cell

# We delete these to avoid name conflicts for automagic to work
del lmagic, lcmagic
```

You can also create magics of all three kinds by inheriting from the `IPython.core.magic.Magics` class. This lets you create magics that can potentially hold state in between calls, and that have full access to the main IPython object:

```
# This code can be put in any Python module, it does not require IPython
# itself to be running already.  It only creates the magics subclass but
# doesn't instantiate it yet.
from IPython.core.magic import (Magics, magics_class, line_magic,
                                cell_magic, line_cell_magic)

# The class MUST call this class decorator at creation time
@magics_class
class MyMagics(Magics):

    @line_magic
    def lmagic(self, line):
        "my line magic"
        print "Full access to the main IPython object:", self.shell
        print "Variables in the user namespace:", self.shell.user_ns.keys()
        return line

    @cell_magic
    def cmagic(self, line, cell):
        "my cell magic"
        return line, cell

    @line_cell_magic
    def lcmagic(self, line, cell=None):
        "Magic that works both as %lcmagic and as %%lcmagic"
        if cell is None:
            print "Called as line magic"
            return line
        else:
            print "Called as cell magic"
            return line, cell


# In order to actually use these magics, you must register them with a
# running IPython.  This code must be placed in a file that is loaded once
# IPython is up and running:
ip = get_ipython()
# You can register the class itself without instantiating it.  IPython will
# call the default constructor on it.
ip.register_magics(MyMagics)
```

If you want to create a class with a different constructor that holds additional state, then you should always call the parent constructor and instantiate the class yourself before registration:

```
@magics_class
class StatefulMagics(Magics):
    "Magics that hold additional state"

    def __init__(self, shell, data):
        # You must call the parent constructor
        super(StatefulMagics, self).__init__(shell)
        self.data = data

    # etc...

# This class must then be registered with a manually created instance,
# since its constructor has different arguments from the default:
ip = get_ipython()
magics = StatefulMagics(ip, some_data)
ip.register_magics(magics)
```

In earlier versions, IPython had an API for the creation of line magics (cell magics did not exist at the time) that required you to create functions with a method-looking signature and to manually pass both the function and the name. While this API is no longer recommended, it remains indefinitely supported for backwards compatibility purposes. With the old API, you'd create a magic as follows:

```
def func(self, line):
    print "Line magic called with line:", line
    print "IPython object:", self.shell

ip = get_ipython()
# Declare this function as the magic %mycommand
ip.define_magic('mycommand', func)
```

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* 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

Simply type `help()` to access Python's standard help system. You can also type `help(object)` for information about a given object, or `help('keyword')` for information on a keyword. You may need to configure your PYTHONDOCS environment variable 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 (e.g. function signatures) they get snipped in the center for brevity. This system gives access variable types and values, docstrings, function prototypes and other useful information.

If the information will not fit in the terminal, it is displayed in a pager (`less` if available, otherwise a basic internal pager).

Typing `??word` or `word??` gives access to the full information, including the source code where possible. Long strings are not snipped.

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 (`%function_name?`); 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 call signature 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`) work on object attributes, as well as directly on variables. For example, after doing `import os`, you can use `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 $IPYTHONDIR/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 you don't already have a ~/.inputrc file, you need this include:
$include /etc/inputrc

$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 in the terminal, 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 configuration file (note that these options can not be specified at the command line):

- **readline_parse_and_bind**: this holds a list of strings 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.

You will find the default values in your configuration 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

You can assign the result of a system command to a Python variable with the syntax `myfiles = !ls`. This gets machine readable output from stdout (e.g. without colours), and splits on newlines. To explicitly get this sort of output without assigning to a variable, use two exclamation marks (`!!ls`) or the `%sx` magic command.

The captured list has some convenience features. `myfiles.n` or `myfiles.s` returns a string delimited by newlines or spaces, respectively. `myfiles.p` produces path objects from the list items. See *String lists* for details.

IPython also allows you to expand the value of python variables when making system calls. Wrap variables or expressions in {braces}:

```
In [1]: pyvar = 'Hello world'
In [2]: !echo "A python variable: {pyvar}"
A python variable: Hello world
In [3]: import math
In [4]: x = 8
In [5]: !echo {math.factorial(x)}
40320
```

For simple cases, you can alternatively prepend $ to a variable name:

```
In [6]: !echo $sys.argv
[/home/fperez/usr/bin/ipython]
In [7]: !echo "A system variable: $$HOME"  # Use $$ for literal $
A system variable: /home/fperez
```

System command aliases

The %alias magic function allows 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
ERROR: Alias <parts> requires 2 arguments, 1 given.
```

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 `IPython.lib.deepreload` module allows you to recursively reload a module: changes made to any of its dependencies will be reloaded without having to exit. To start using it, do:

```
from IPython.lib.deepreload import reload as dreload
```

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 %rerun or %macro functions. 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 variables are also 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 configuration 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):

```
In [1]: callable_ob arg1, arg2, arg3
------> callable_ob(arg1, arg2, arg3)
```

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

```
In [2]: /globals # becomes 'globals()'
```

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

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

but this will work:

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

```
In [1]: ,my_function /home/me  # becomes my_function("/home/me")
```

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

```
In [2]: ,my_function a b c    # becomes my_function("a","b","c")

In [3]: ;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:

```
In [4]: 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 the following code at the end of that file, then IPython will be your working environment anytime you start Python:

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

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

You can start a regular IPython session with

```
import IPython
IPython.start_ipython()
```

at any point in your program. This will load IPython configuration, startup files, and everything, just as if it were a normal IPython session. In addition to this, it is possible to embed 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.

Note: At present, embedding IPython cannot be done from inside IPython. Run the code samples below outside IPython.

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
```

Note: As of 0.13, you can embed an IPython *kernel*, for use with qtconsole, etc. via `IPython.embed_kernel()` instead of `IPython.embed()`. It should function just the same as regular embed, but you connect an external frontend rather than IPython starting up in the local terminal.

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."""
from __future__ import print_function

# 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()
    prompt_config = cfg.PromptManager
    prompt_config.in_template = 'In <\\#>: '
    prompt_config.in2_template = '   .\\D.: '
    prompt_config.out_template = '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.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()
prompt_config = cfg2.PromptManager
prompt_config.in_template = 'In2<\\#>: '
if not nested:
    prompt_config.in_template = 'In2<\\#>: '
    prompt_config.in2_template = '   .\\D.: '
    prompt_config.out_template = '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.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.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 config 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 pdb will automatically 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`.

IPython has excellent support for working interactively with Graphical User Interface (GUI) toolkits, such as wxPython, PyQt4/PySide, 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`, `qt`, `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 `matplotlib` 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. The integration with various eventloops is handled somewhat differently (and more simply) when using the standalone kernel, as in the qtconsole and notebook.

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 detects 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 `examples/lib` that demonstrate these capabilities.

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 `examples/lib` that demonstrate these capabilities.

PyQt and PySide

When you use `--gui=qt` or `--matplotlib=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 matplotlib mode with `ipython --matplotlib=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 `--matplotlib` switch. If IPython is already running, you can run the `%matplotlib` magic. If no arguments are given, IPython will automatically detect your choice of matplotlib backend. You can also request a specific backend with `%matplotlib backend`, where `backend` must be one of: 'tk', 'qt', 'wx', 'gtk', 'osx'. In the web notebook and Qt console, 'inline' is also a valid backend value, which produces static figures inlined inside the application window instead of matplotlib's interactive figures that live in separate windows.

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:

```
# -*- coding: utf-8 -*-
"""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.

This is a unicode test, åäö
"""
from __future__ import print_function

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 cannot break up sections within 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

4.4.1 Overview

It is possible to adapt IPython for system shell usage. In the past, IPython shipped a special 'sh' profile for this purpose, but it had been quarantined since 0.11 release, and in 1.0 it was removed altogether. Nevertheless, much of this section relies on machinery which does not require a custom profile.

You can set up your own 'sh' *profile* to be different from the default profile such that:

- Prompt shows the current directory (see *Prompt customization*)
- Make system commands directly available (in alias table) by running the `%rehashx` magic. If you install new programs along your PATH, you might want to run `%rehashx` to update the alias table
- turn `%autocall` to full mode

4.4.2 Aliases

Once you run `%rehashx`, 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 Environment variables

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

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

4.4.5 Prompt customization

Here are some prompt configurations you can try out interactively by using the `%config` magic:

```
%config PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}[{color.Light
%config PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
%config PromptManager.out_template = r'<\#> '
```

You can change the prompt configuration to your liking permanently by editing `ipython_config.py`:

```
c.PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}[{color.LightCyan}
c.PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
c.PromptManager.out_template = r'<\#> '
```

Read more about the *configuration system* for details on how to find `ipython_config.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 if files), use the .s property:

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

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’.

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 a newline (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.

interactive/../../_images/qtconsole.png

Fig. 4.1: The Qt console for IPython, using inline matplotlib plots.

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 `Ctrl-Enter` at the end of the first line instead of `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 `Shift-Enter`.

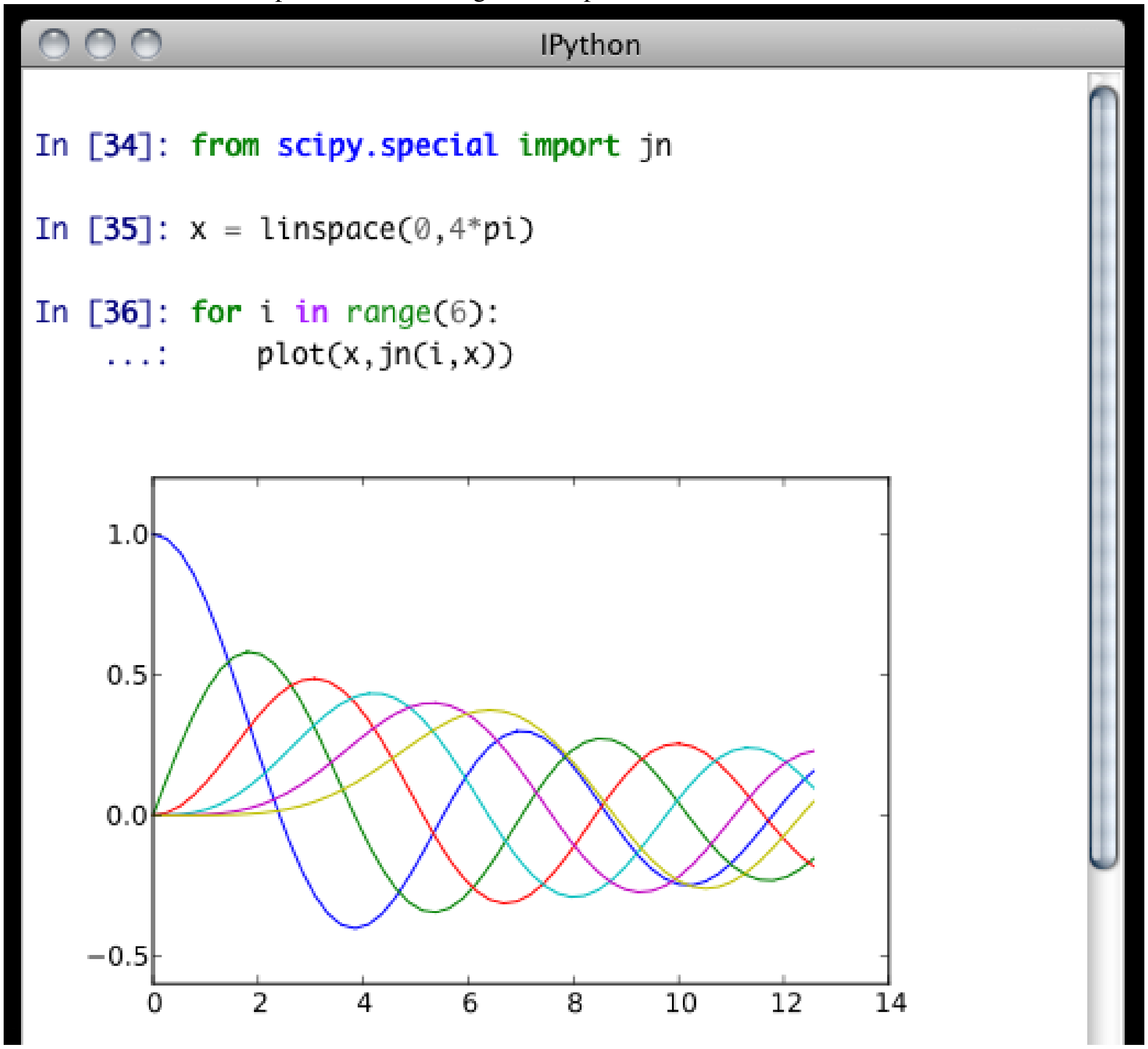
4.5.1 `%load`

The new `%load` magic (previously `%loadpy`) takes any script, 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 an history range, or 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]: %load http://matplotlib.org/plot_directive/mpl_examples/mplot3d/contour3d_demo.py

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

4.5.2 Inline Matplotlib

One of the most exciting features of the QtConsole is embedded matplotlib figures. You can use any standard matplotlib GUI backend 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.



```
In [34]: from scipy.special import jn

In [35]: x = linspace(0,4*pi)

In [36]: for i in range(6):
    ...:     plot(x,jn(i,x))
```

display()

IPython provides a function `display()` for displaying rich representations of objects if they are available. The IPython display system provides a mechanism for specifying PNG or SVG (and more) representations of objects for GUI frontends. When you enable matplotlib integration via the `%matplotlib` magic, 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 [4]: from IPython.display import display

In [5]: plt.plot(range(5)) # plots in the matplotlib window

In [6]: display(plt.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 = plt.figure()

In [2]: plt.plot(np.rand(100))
Out[2]: [<matplotlib.lines.Line2D at 0x7fc6ac03dd90>]

In [3]: display(f)

# Plot is shown here

In [4]: plt.title('A title')
Out[4]: <matplotlib.text.Text at 0x7fc6ac023450>

In [5]: display(f)

# Updated plot with title is shown here.
```

`--matplotlib inline`

If you want to have all of your figures embedded in your session, instead of calling `display()`, you can specify `--matplotlib 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)()`.

The inline backend can use either SVG or PNG figures (PNG being the default). It also supports the special key `'retina'`, which is 2x PNG for high-DPI displays. To switch between them, set the `InlineBackend.figure_format` configurable in a config file, or via the `%config` magic:

```
In [10]: %config InlineBackend.figure_format = 'svg'
```

Note: Changing the inline figure format also affects calls to `display()` above, even if you are not using the inline backend for all figures.

By default, IPython closes all figures at the completion of each execution. This means you don't have to manually close figures, which is less convenient when figures aren't attached to windows with an obvious close button. It also means that the first matplotlib call in each cell will always create a new figure:

```
In [11]: plt.plot(range(100))
<single-line plot>

In [12]: plt.plot([1,3,2])
<another single-line plot>
```

However, it does prevent the list of active figures surviving from one input cell to the next, so if you want to continue working with a figure, you must hold on to a reference to it:

```
In [11]: fig = gcf()
   ....: fig.plot(rand(100))
<plot>
In [12]: fig.title('Random Title')
<redraw plot with title>
```

This behavior is controlled by the `InlineBackend.close_figures` configurable, and if you set it to False, via %config or config file, then IPython will *not* close figures, and tools like `gcf()`, `gca()`, `getfigs()` will behave the same as they do with other backends. You will, however, have to manually close figures:

```
# close all active figures:
In [13]: [ fig.close() for fig in getfigs() ]
```

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* matplotlib, 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.

Note: Export to SVG+XHTML requires that you are using SVG figures, which is *not* the default. To switch the inline figure format to use SVG during an active session, do:

```
In [10]: %config InlineBackend.figure_format = 'svg'
```

Or, you can add the same line (c.Inline... instead of %config Inline...) to your config files.

This will only affect figures plotted after making this call

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:

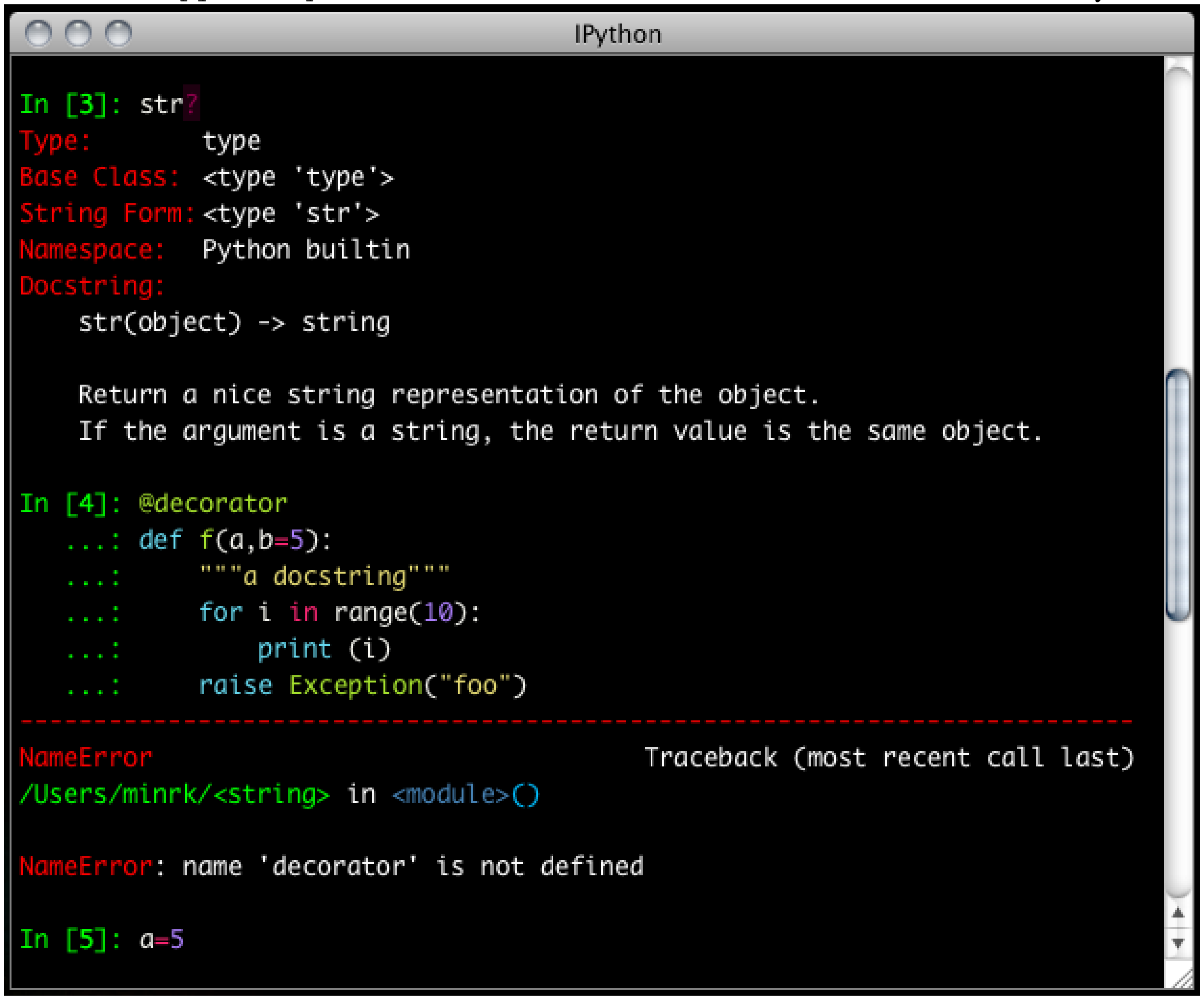
- LightBG for light backgrounds
- Linux for dark backgrounds

- 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:



```
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; }
/* .inverted is used to highlight selected completion */
.inverted { background-color: black ; color: white; }
```

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=
```

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 kernel-12345.json
```

Other frontends can connect to your kernel, and share in the execution. This is great for collaboration. The `--existing` flag means connect to a kernel that already exists. Starting other consoles with that flag will not try to start their own kernel, but rather connect to yours. `kernel-12345.json` is a small JSON file with the ip, port, and authentication information necessary to connect to your kernel. By default, this file will be in your default profile's security directory. If it is somewhere else, the output line will print the full path of the connection file, rather than just its filename.

If you need to find the connection info to send, and don't know where your connection file lives, there are a couple of ways to get it. If you are already running an IPython console connected to the kernel, you can use the `%connect_info` magic to display the information necessary to connect another frontend to the kernel.

```
In [2]: %connect_info
{
  "stdin_port":50255,
  "ip":"127.0.0.1",
  "hb_port":50256,
  "key":"70be6f0f-1564-4218-8cda-31be40a4d6aa",
  "shell_port":50253,
  "iopub_port":50254
}

Paste the above JSON into a file, and connect with:
    $> ipython <app> --existing <file>
or, if you are local, you can connect with just:
    $> ipython <app> --existing kernel-12345.json
or even just:
    $> ipython <app> --existing
if this is the most recent IPython session you have started.
```

Otherwise, you can find a connection file by name (and optionally profile) with `IPython.lib.kernel.find_connection_file()`:

```
$> python -c "from IPython.lib.kernel import find_connection_file;\
print find_connection_file('kernel-12345.json')"
/home/you/.ipython/profile_default/security/kernel-12345.json
```

And if you are using a particular IPython profile:

```
$> python -c "from IPython.lib.kernel import find_connection_file;\
print find_connection_file('kernel-12345.json', profile='foo')"
/home/you/.ipython/profile_foo/security/kernel-12345.json
```

You can even launch a standalone kernel, and connect and disconnect Qt Consoles from various machines. This lets you keep the same running IPython session on your work machine (with matplotlib plots and everything), logging in from home, cafés, etc.:

```
$> ipython kernel
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing kernel-12345.json
```

This is actually exactly the same as the subprocess launched by the qtconsole, so all the information about connecting to a standalone kernel is identical to that of connecting to the kernel attached to a running console.

Security

Warning: Since the ZMQ code currently has no encryption, listening on an external-facing IP is dangerous. You are giving any computer that can see you on the network the ability to connect to your kernel, and view your traffic. Read the rest of this section before listening on external ports or running an IPython kernel on a shared machine.

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 or '*', that means all interfaces, so any computer that can see yours on the network can connect to the kernel.

Messages are not encrypted, so users with access to the ports your kernel is using will be able to see any output of the kernel. They will **NOT** be able to issue shell commands as you due to message signatures, which are enabled by default as of IPython 0.12.

Warning: If you disable message signatures, then any user with access to the ports your kernel is listening on can issue arbitrary code as you. **DO NOT** disable message signatures unless you have a lot of trust in your environment.

The one security feature IPython does provide is protection from unauthorized execution. IPython's messaging system will sign messages with HMAC digests using a shared-key. The key is never sent over the network, it is only used to generate a unique hash for each message, based on its content. When IPython receives a message, it will check that the digest matches, and discard the message. You can use any file that only you have access to to generate this key, but the default is just to generate a new UUID. You can generate a random private key with:

```
# generate 1024b of random data, and store in a file only you can read:
# (assumes IPYTHONDIR is defined, otherwise use your IPython directory)
$> python -c "import os; print os.urandom(128).encode('base64')" > $IPYTHONDIR/sessionkey
$> chmod 600 $IPYTHONDIR/sessionkey
```

The *contents* of this file will be stored in the JSON connection file, so that file contains everything you need to connect to and use a kernel.

To use this generated key, simply specify the `Session.keyfile` configurable in `ipython_config.py` or at the command-line, as in:

```
# instruct IPython to sign messages with that key, instead of a new UUID
$> ipython qtconsole --Session.keyfile=$IPYTHONDIR/sessionkey
```

SSH Tunnels

Sometimes you want to connect to machines across the internet, or just across a LAN that either doesn't permit open ports or you don't trust the other machines on the network. To do this, you can use SSH tunnels. SSH tunnels are a way to securely forward ports on your local machine to ports on another machine, to which you have SSH access.

In simple cases, IPython's tools can forward ports over ssh by simply adding the `--ssh=remote` argument to the usual `--existing...` set of flags for connecting to a running kernel, after copying the JSON connection file (or its contents) to the second computer.

Warning: Using SSH tunnels does *not* increase localhost security. In fact, when tunneling from one machine to another *both* machines have open ports on localhost available for connections to the kernel.

There are two primary models for using SSH tunnels with IPython. The first is to have the Kernel listen only on localhost, and connect to it from another machine on the same LAN.

First, let's start a kernel on machine **worker**, listening only on loopback:

```
user@worker $> ipython kernel
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing kernel-12345.json
```

In this case, the IP that you would connect to would still be 127.0.0.1, but you want to specify the additional `--ssh` argument with the hostname of the kernel (in this example, it's 'worker'):

```
user@client $> ipython qtconsole  --ssh=worker --existing /path/to/kernel-12345.json
```

Which will write a new connection file with the forwarded ports, so you can reuse them:

```
[IPythonQtConsoleApp] To connect another client via this tunnel, use:
[IPythonQtConsoleApp] --existing kernel-12345-ssh.json
```

Note again that this opens ports on the *client* machine that point to your kernel.

Note: the ssh argument is simply passed to openssh, so it can be fully specified `user@host:port` but it will also respect your aliases, etc. in `.ssh/config` if you have any.

The second pattern is for connecting to a machine behind a firewall across the internet (or otherwise wide network). This time, we have a machine **login** that you have ssh access to, which can see **kernel**, but **client** is on another network. The important difference now is that **client** can see **login**, but *not* **worker**. So we need to forward ports from client to worker *via* login. This means that the kernel must be started listening on external interfaces, so that its ports are visible to *login*:

```
user@worker $> ipython kernel --ip=0.0.0.0
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing kernel-12345.json
```

Which we can connect to from the client with:

```
user@client $> ipython qtconsole --ssh=login --ip=192.168.1.123 --existing /path/to/kernel-
```

Note: The IP here is the address of worker as seen from *login*, and need only be specified if the kernel used the ambiguous 0.0.0.0 (all interfaces) address. If it had used 192.168.1.123 to start with, it would not be needed.

Manual SSH tunnels

It's possible that IPython's ssh helper functions won't work for you, for various reasons. You can still connect to remote machines, as long as you set up the tunnels yourself. The basic format of forwarding a local port to a remote one is:

```
[client] $> ssh <server> <localport>:<remoteip>:<remoteport> -f -N
```

This will forward local connections to **localport** on client to **remoteip:remoteport** *via* **server**. Note that remoteip is interpreted relative to *server*, not the client. So if you have direct ssh access to the machine to which you want to forward connections, then the server *is* the remote machine, and remoteip should be server's IP as seen from the server itself, i.e. 127.0.0.1. Thus, to forward local port 12345 to remote port 54321 on a machine you can see, do:

```
[client] $> ssh machine 12345:127.0.0.1:54321 -f -N
```

But if your target is actually on a LAN at 192.168.1.123, behind another machine called **login**, then you would do:

```
[client] $> ssh login 12345:192.168.1.16:54321 -f -N
```

The `-f -N` on the end are flags that tell ssh to run in the background, and don't actually run any commands beyond creating the tunnel.

See also:

A short discussion of ssh tunnels: http://www.revsys.com/writings/quicktips/ssh-tunnel.html

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 `--matplotlib 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.

4.6 The IPython Notebook

4.6.1 Introduction

The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The IPython notebook combines two components:

A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.

Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

See also:

See the *installation documentation* for directions on how to install the notebook and its dependencies.

Main features of the web application

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

Notebook documents

Notebook documents contains the inputs and outputs of a interactive session as well as additional text that accompanies the code but is not meant for execution. In this way, notebook files can serve as a complete computational record of a session, interleaving executable code with explanatory text, mathematics, and rich representations of resulting objects. These documents are internally JSON files and are saved with the `.ipynb` extension. Since JSON is a plain text format, they can be version-controlled and shared with colleagues.

Notebooks may be exported to a range of static formats, including HTML (for example, for blog posts), reStructeredText, LaTeX, PDF, and slide shows, via the new *nbconvert* command.

Furthermore, any `.ipynb` notebook document available from a public URL can be shared via the IPython Notebook Viewer (nbviewer). This service loads the notebook document from the URL and renders it as a static web page. The results may thus be shared with a colleague, or as a public blog post, without other users needing to install IPython themselves. In effect, nbviewer is simply *nbconvert* as a web service, so you can do your own static conversions with nbconvert, without relying on nbviewer.

See also:

Details on the notebook JSON file format

4.6.2 Starting the notebook server

You can start running a notebook server from the command line using the following command:

```
ipython notebook
```

This will print some information about the notebook server in your console, and open a web browser to the URL of the web application (by default, `http://127.0.0.1:8888`).

The landing page of the IPython notebook web application, the **dashboard**, shows the notebooks currently available in the notebook directory (by default, the directory from which the notebook server was started).

You can create new notebooks from the dashboard with the `New Notebook` button, or open existing ones by clicking on their name. You can also drag and drop `.ipynb` notebooks and standard `.py` Python source code files into the notebook list area.

When starting a notebook server from the command line, you can also open a particular notebook directly, bypassing the dashboard, with `ipython notebook my_notebook.ipynb`. The `.ipynb` extension is assumed if no extension is given.

When you are inside an open notebook, the *File | Open...* menu option will open the dashboard in a new browser tab, to allow you to open another notebook from the notebook directory or to create a new notebook.

Note: You can start more than one notebook server at the same time, if you want to work on notebooks in different directories. By default the first notebook server starts on port 8888, and later notebook servers search for ports near that one. You can also manually specify the port with the `--port` option.

Creating a new notebook document

A new notebook may be created at any time, either from the dashboard, or using the *File | New* menu option from within an active notebook. The new notebook is created within the same directory and will open in a new browser tab. It will also be reflected as a new entry in the notebook list on the dashboard.

Opening notebooks

An open notebook has **exactly one** interactive session connected to an *IPython kernel*, which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel. In the dashboard, notebooks with an active kernel have a `Shutdown` button next to them, whereas notebooks without an active kernel have a `Delete` button in its place.

Other clients may connect to the same underlying IPython kernel. The notebook server always prints to the terminal the full details of how to connect to each kernel, with messages such as the following:

```
[NotebookApp] Kernel started: 87f7d2c0-13e3-43df-8bb8-1bd37aaf3373
```

This long string is the kernel's ID which is sufficient for getting the information necessary to connect to the kernel. You can also request this connection data by running the `%connect_info` *magic*. This will print the same ID information as well as the content of the JSON data structure it contains.

You can then, for example, manually start a Qt console connected to the *same* kernel from the command line, by passing a portion of the ID:

```
$ ipython qtconsole --existing 87f7d2c0
```

Without an ID, `--existing` will connect to the most recently started kernel. This can also be done by running the `%qtconsole` *magic* in the notebook.

See also:

Decoupled two-process model

4.6.3 Notebook user interface

When you create a new notebook document, you will be presented with the **notebook name**, a **menu bar**, a **toolbar** and an empty **code cell**.

notebook name: The name of the notebook document is displayed at the top of the page, next to the `IP[y]: Notebook` logo. This name reflects the name of the `.ipynb` notebook document file. Clicking on the notebook name brings up a dialog which allows you to rename it. Thus, renaming a notebook from "Untitled0" to "My first notebook" in the browser, renames the `Untitled0.ipynb` file to `My first notebook.ipynb`.

menu bar: The menu bar presents different options that may be used to manipulate the way the notebook functions.

toolbar: The tool bar gives a quick way of performing the most-used operations within the notebook, by clicking on an icon.

code cell: the default type of cell, read on for an explanation of cells

4.6.4 Structure of a notebook document

The notebook consists of a sequence of cells. A cell is a multi-line text input field, and its contents can be executed by using `Shift-Enter`, or by clicking either the "Play" button the toolbar, or *Cell | Run* in the menu bar. The execution behavior of a cell is determined the cell's type. There are four types of cells: **code cells**, **markdown cells**, **raw cells** and **heading cells**. Every cell starts off being a **code cell**, but its type can be changed by using a dropdown on the toolbar (which will be "Code", initially), or via *keyboard shortcuts*.

Code cells

A *code cell* allows you to edit and write new code, with full syntax highlighting and tab completion. By default, the language associated to a code cell is Python, but other languages, such as `Julia` and `R`, can be handled using *cell magic commands*.

When a code cell is executed, code that it contains is sent to the kernel associated with the notebook. The results that are returned from this computation are then displayed in the notebook as the cell's *output*. The output is not limited to text, with many other possible forms of output are also possible, including `matplotlib` figures and HTML tables (as used, for example, in the `pandas` data analysis package). This is known as IPython's *rich display* capability.

See also:

Basic Output example notebook

Rich Display System example notebook

Markdown cells

You can document the computational process in a literate way, alternating descriptive text with code, using *rich text*. In IPython this is accomplished by marking up text with the Markdown language. The corresponding cells are called *Markdown cells*. The Markdown language provides a simple way to perform this text markup, that is, to specify which parts of the text should be emphasized (italics), bold, form lists, etc.

When a Markdown cell is executed, the Markdown code is converted into the corresponding formatted rich text. Markdown allows arbitrary HTML code for formatting.

Within Markdown cells, you can also include *mathematics* in a straightforward way, using standard LaTeX notation: `$...$` for inline mathematics and `$$...$$` for displayed mathematics. When the Markdown cell is executed, the LaTeX portions are automatically rendered in the HTML output as equations with high quality typography. This is made possible by MathJax, which supports a large subset of LaTeX functionality

Standard mathematics environments defined by LaTeX and AMS-LaTeX (the *amsmath* package) also work, such as `\begin{equation}...\end{equation}`, and `\begin{align}...\end{align}`. New LaTeX macros may be defined using standard methods, such as `\newcommand`, by placing them anywhere *between math delimiters* in a Markdown cell. These definitions are then available throughout the rest of the IPython session.

See also:

Markdown Cells example notebook

Raw cells

Raw cells provide a place in which you can write *output* directly. Raw cells are not evaluated by the notebook. When passed through *nbconvert*, raw cells arrive in the destination format unmodified. For example, this allows you to type full LaTeX into a raw cell, which will only be rendered by LaTeX after conversion by nbconvert.

Heading cells

You can provide a conceptual structure for your computational document as a whole using different levels of headings; there are 6 levels available, from level 1 (top level) down to level 6 (paragraph). These can be used later for constructing tables of contents, etc. As with Markdown cells, a heading cell is replaced by a rich text rendering of the heading when the cell is executed.

4.6.5 Basic workflow

The normal workflow in a notebook is, then, quite similar to a standard IPython session, with the difference that you can edit cells in-place multiple times until you obtain the desired results, rather than having to rerun separate scripts with the `%run` magic command.

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

At certain moments, it may be necessary to interrupt a calculation which is taking too long to complete. This may be done with the *Kernel | Interrupt* menu option, or the `Ctrl-m i` keyboard shortcut. Similarly, it may be necessary or desirable to restart the whole computational process, with the *Kernel | Restart* menu option or `Ctrl-m .` shortcut.

A notebook may be downloaded in either a `.ipynb` or `.py` file from the menu option *File | Download as*. Choosing the `.py` option downloads a Python `.py` script, in which all rich output has been removed and the content of markdown cells have been inserted as comments.

See also:

Running Code in the IPython Notebook example notebook

Basic Output example notebook

a warning about doing "roundtrip" conversions.

Keyboard shortcuts

All actions in the notebook can be performed with the mouse, but keyboard shortcuts are also available for the most common ones. The essential shortcuts to remember are the following:

- **`Shift-Enter`: run cell** Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter` is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.
- **`Ctrl-Enter`: run cell in-place** Execute the current cell as if it were in "terminal mode", where any output is shown, but the cursor *remains* in the current cell. The cell's entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.
- **`Alt-Enter`: run cell, insert below** Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). This is thus a shortcut for the sequence `Shift-Enter`, `Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.)
- `Ctrl-m`: This is the prefix for *all* other shortcuts, which consist of `Ctrl-m` followed by a single letter or character. For example, if you type `Ctrl-m h` (that is, the sole letter `h` after `Ctrl-m`), IPython will show you all the available keyboard shortcuts.

Here is the complete set of keyboard shortcuts available:

Shortcut	**Action**
Shift-Enter	run cell
Ctrl-Enter	run cell in-place
Alt-Enter	run cell, insert below
Ctrl-m x	cut cell
Ctrl-m c	copy cell
Ctrl-m v	paste cell
Ctrl-m d	delete cell
Ctrl-m z	undo last cell deletion
Ctrl-m -	split cell
Ctrl-m a	insert cell above
Ctrl-m b	insert cell below
Ctrl-m o	toggle output
Ctrl-m O	toggle output scroll
Ctrl-m l	toggle line numbers
Ctrl-m s	save notebook
Ctrl-m j	move cell down
Ctrl-m k	move cell up
Ctrl-m y	code cell
Ctrl-m m	markdown cell
Ctrl-m t	raw cell
Ctrl-m 1-6	heading 1-6 cell
Ctrl-m p	select previous
Ctrl-m n	select next
Ctrl-m i	interrupt kernel
Ctrl-m .	restart kernel
Ctrl-m h	show keyboard shortcuts

4.6.6 Plotting

One major feature of the notebook is the ability to display plots that are the output of running code cells. IPython is designed to work seamlessly with the matplotlib plotting library to provide this functionality.

To set this up, before any plotting is performed you must execute the `%matplotlib` *magic command*. This performs the necessary behind-the-scenes setup for IPython to work correctly hand in hand with `matplotlib`; it does *not*, however, actually execute any Python `import` commands, that is, no names are added to the namespace.

If the `%matplotlib` magic is called without an argument, the output of a plotting command is displayed using the default `matplotlib` backend in a separate window. Alternatively, the backend can be explicitly requested using, for example:

```
%matplotlib gtk
```

A particularly interesting backend, provided by IPython, is the `inline` backend. This is available only for the IPython Notebook and the *IPython QtConsole*. It can be invoked as follows:

```
%matplotlib inline
```

With this backend, the output of plotting commands is displayed *inline* within the notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.

See also:

Plotting with Matplotlib example notebook

4.6.7 Configuring the IPython Notebook

The notebook server can be run with a variety of command line arguments. To see a list of available options enter:

```
$ ipython notebook --help
```

Defaults for these options can also be set by creating a file named `ipython_notebook_config.py` in your IPython *profile folder*. The profile folder is a subfolder of your IPython directory; to find out where it is located, run:

```
$ ipython locate
```

To create a new set of default configuration files, with lots of information on available options, use:

```
$ ipython profile create
```

See also:

Overview of the IPython configuration system, in particular *Profiles*.

Notebook security

Running a public notebook server

4.6.8 Importing `.py` files

`.py` files will be imported as a notebook with the same basename, but an `.ipynb` extension, located in the notebook directory. The notebook created will have just one cell, which will contain all the code in the `.py` file. You can later manually partition this into individual cells using the `Edit | Split Cell` menu option, or the `Ctrl-m -` keyboard shortcut.

Note that `.py` scripts obtained from a notebook document using **nbconvert_** maintain the structure of the notebook in comments. Reimporting such a script back into a notebook will preserve this structure.

Warning: While in simple cases you can "roundtrip" a notebook to Python, edit the Python file, and then import it back without loss of main content, this is in general *not guaranteed to work*. First, there is extra metadata saved in the notebook that may not be saved to the `.py` format. And as the notebook format evolves in complexity, there will be attributes of the notebook that will not survive a roundtrip through the Python form. You should think of the Python format as a way to output a script version of a notebook and the import capabilities as a way to load existing code to get a notebook started. But the Python version is *not* an alternate notebook format.

See also:

Notebook JSON file format

4.7 Converting notebooks to other formats

Newly added in the 1.0 release of IPython is the `nbconvert` tool, which allows you to convert an `.ipynb` notebook document file into various static formats.

Currently, `nbconvert` is provided as a command line tool, run as a script using IPython. A direct export capability from within the IPython Notebook web app is planned.

The command-line syntax to run the `nbconvert` script is:

```
$ ipython nbconvert --to FORMAT notebook.ipynb
```

This will convert the IPython document file `notebook.ipynb` into the output format given by the `FORMAT` string.

The default output format is html, for which the `--to` argument may be omitted:

```
$ ipython nbconvert notebook.ipynb
```

IPython provides a few templates for some output formats, and these can be specified via an additional `--template` argument.

The currently supported export formats are:

- `--to html`
  - `--template full` (default)

    A full static HTML render of the notebook. This looks very similar to the interactive view.
  - `--template basic`

    Simplified HTML, useful for embedding in webpages, blogs, etc. This excludes HTML headers.
- `--to latex`

  Latex export. This generates `NOTEBOOK_NAME.tex` file, ready for export. You can automatically run latex on it to generate a PDF by adding `--post PDF`.
  - `--template article` (default)

    Latex article, derived from Sphinx's howto template.
  - `--template book`

    Latex book, derived from Sphinx's manual template.
  - `--template basic`

    Very basic latex output - mainly meant as a starting point for custom templates.

- `--to slides`

  This generates a Reveal.js HTML slideshow. It must be served by an HTTP server. The easiest way to get this is to add `--post serve` on the command-line.

- `--to markdown`

  Simple markdown output. Markdown cells are unaffected, and code cells are placed in triple-backtick (` ``` `) blocks.

- `--to rst`

  Basic reStructuredText output. Useful as a starting point for embedding notebooks in Sphinx docs.

- `--to python`

  Convert a notebook to an executable Python script. This is the simplest way to get a Python script out of a notebook. If there were any magics in the notebook, this may only be executable from an IPython session.

Note: nbconvert uses pandoc to convert between various markup languages, so pandoc is a dependency of most nbconvert transforms, excluding Markdown and Python.

The output file created by `nbconvert` will have the same base name as the notebook and will be placed in the current working directory. Any supporting files (graphics, etc) will be placed in a new directory with the same base name as the notebook, suffixed with `_files`:

```
$ ipython nbconvert notebook.ipynb
$ ls
notebook.ipynb   notebook.html    notebook_files/
```

For simple single-file output, such as html, markdown, etc., the output may be sent to standard output with:

```
$ ipython nbconvert --to markdown notebook.ipynb --stdout
```

Multiple notebooks can be specified from the command line:

```
$ ipython nbconvert notebook*.ipynb
$ ipython nbconvert notebook1.ipynb notebook2.ipynb
```

or via a list in a configuration file, say `mycfg.py`, containing the text:

```
c = get_config()
c.NbConvertApp.notebooks = ["notebook1.ipynb", "notebook2.ipynb"]
```

and using the command:

```
$ ipython nbconvert --config mycfg.py
```

4.7.1 Notebook JSON file format

Notebook documents are JSON files with an `.ipynb` extension, formatted as legibly as possible with minimal extra indentation and cell content broken across lines to make them reasonably friendly to use in version-control workflows. You should be very careful if you ever manually edit this JSON data, as it is extremely easy to corrupt its internal structure and make the file impossible to load. In general, you should consider the notebook as a file meant only to be edited by the IPython Notebook app itself, not for hand-editing.

Note: Binary data such as figures are also saved directly in the JSON file. This provides convenient single-file portability, but means that the files can be large; a `diff` of binary data is also not very meaningful. Since the binary blobs are encoded in a single line, they affect only one line of the `diff` output, but they are typically very long lines. You can use the `Cell | All Output | Clear` menu option to remove all output from a notebook prior to committing it to version control, if this is a concern.

The notebook server can also generate a pure Python version of your notebook, using the `File | Download as` menu option. The resulting `.py` file will contain all the code cells from your notebook verbatim, and all Markdown cells prepended with a comment marker. The separation between code and Markdown cells is indicated with special comments and there is a header indicating the format version. All output is removed when exporting to Python.

As an example, consider a simple notebook called `simple.ipynb` which contains one Markdown cell, with the content `The simplest notebook.`, one code input cell with the content `print "Hello, IPython!"`, and the corresponding output.

The contents of the notebook document `simple.ipynb` is the following JSON container:

```
{
 "metadata": {
  "name": "simple"
 },
 "nbformat": 3,
 "nbformat_minor": 0,
 "worksheets": [
  {
   "cells": [
    {
     "cell_type": "markdown",
     "metadata": {},
     "source": "The simplest notebook."
    },
    {
     "cell_type": "code",
     "collapsed": false,
     "input": "print \"Hello, IPython\"",
     "language": "python",
     "metadata": {},
     "outputs": [
      {
       "output_type": "stream",
       "stream": "stdout",
       "text": "Hello, IPython\n"
      }
     ],
     "prompt_number": 1
    }
   ],
   "metadata": {}
  }
 ]
}
```

The corresponding Python script is:

```
# -*- coding: utf-8 -*-
# <nbformat>3.0</nbformat>

# <markdowncell>

# The simplest notebook.

# <codecell>

print "Hello, IPython"
```

Note that indeed the output of the code cell, which is present in the JSON container, has been removed in the `.py` script.

4.8 Running a notebook server

The *IPython notebook* web-application is based on a server-client structure. This server uses a *two-process kernel architecture* based on ZeroMQ, as well as Tornado for serving HTTP requests. By default, a notebook server runs on http://127.0.0.1:8888/ and is accessible only from *localhost*. This document describes how you can *secure a notebook server* and how to *run it on a public interface*.

4.8.1 Notebook security

You can protect your notebook server with a simple single password by setting the `NotebookApp.password` configurable. You can prepare a hashed password using the function `IPython.lib.security.passwd()`:

```
In [1]: from IPython.lib import passwd
In [2]: passwd()
Enter password:
Verify password:
Out[2]: 'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

Note: `passwd()` can also take the password as a string argument. **Do not** pass it as an argument inside an IPython session, as it will be saved in your input history.

You can then add this to your `ipython_notebook_config.py`, e.g.:

```
# Password to use for web authentication
c = get_config()
```

```
c.NotebookApp.password =
u'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

When using a password, it is a good idea to also use SSL, so that your password is not sent unencrypted by your browser. You can start the notebook to communicate via a secure protocol mode using a self-signed certificate with the command:

```
$ ipython notebook --certfile=mycert.pem
```

Note: A self-signed certificate can be generated with `openssl`. For example, the following command will create a certificate valid for 365 days with both the key and certificate data written to the same file:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mycert.pem -out mycert.pem
```

Your browser will warn you of a dangerous certificate because it is self-signed. If you want to have a fully compliant certificate that will not raise warnings, it is possible (but rather involved) to obtain one, as explained in detail in this tutorial.

Keep in mind that when you enable SSL support, you will need to access the notebook server over `https://`, not over plain `http://`. The startup message from the server prints this, but it is easy to overlook and think the server is for some reason non-responsive.

4.8.2 Running a public notebook server

If you want to access your notebook server remotely via a web browser, you can do the following.

Start by creating a certificate file and a hashed password, as explained above. Then create a custom profile for the notebook, with the following command line, type:

```
$ ipython profile create nbserver
```

In the profile directory just created, edit the file `ipython_notebook_config.py`. By default, the file has all fields commented; the minimum set you need to uncomment and edit is the following:

```
c = get_config()

# Kernel config
c.IPKernelApp.pylab = 'inline'  # if you want plotting support always

# Notebook config
c.NotebookApp.certfile = u'/absolute/path/to/your/certificate/mycert.pem'
c.NotebookApp.ip = '*'
c.NotebookApp.open_browser = False
c.NotebookApp.password = u'sha1:bcd259ccf...[your hashed password here]'
# It is a good idea to put it on a known, fixed port
c.NotebookApp.port = 9999
```

You can then start the notebook and access it later by pointing your browser to `https://your.host.com:9999` with `ipython notebook --profile=nbserver`.

4.8.3 Running with a different URL prefix

The notebook dashboard (the landing page with an overview of the notebooks in your working directory) typically lives at the URL `http://localhost:8888/`. If you prefer that it lives, together with the rest of the notebook, under a sub-directory, e.g. `http://localhost:8888/ipython/`, you can do so with configuration options like the following (see above for instructions about modifying `ipython_notebook_config.py`):

```
c.NotebookApp.base_project_url = '/ipython/'
c.NotebookApp.base_kernel_url = '/ipython/'
c.NotebookApp.webapp_settings = {'static_url_prefix':'/ipython/static/'}
```

4.8.4 Using a different notebook store

By default, the notebook server stores the notebook documents that it saves as files in the working directory of the notebook server, also known as the `notebook_dir`. This logic is implemented in the `FileNotebookManager` class. However, the server can be configured to use a different notebook manager class, which can store the notebooks in a different format.

Currently, we ship a `AzureNotebookManager` class that stores notebooks in Azure blob storage. This can be used by adding the following lines to your `ipython_notebook_config.py` file:

```
c.NotebookApp.notebook_manager_class =
'IPython.html.services.notebooks.azurenbmanager.AzureNotebookManager'
c.AzureNotebookManager.account_name = u'paste_your_account_name_here'
c.AzureNotebookManager.account_key = u'paste_your_account_key_here'
c.AzureNotebookManager.container = u'notebooks'
```

In addition to providing your Azure Blob Storage account name and key, you will have to provide a container name; you can use multiple containers to organize your notebooks.

4.8.5 Known issues

When behind a proxy, especially if your system or browser is set to autodetect the proxy, the notebook web application might fail to connect to the server's websockets, and present you with a warning at startup. In this case, you need to configure your system not to use the proxy for the server's address.

For example, in Firefox, go to the Preferences panel, Advanced section, Network tab, click 'Settings...', and add the address of the notebook server to the 'No proxy for' field.

CHAPTER 5

Using IPython for parallel computing

5.1 Overview and getting started

5.1.1 Examples

We have various example scripts and notebooks for using IPython.parallel in our `examples/parallel` directory, or they can be found on GitHub. Some of these are covered in more detail in the *examples* section.

5.1.2 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.

5.1.3 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*.

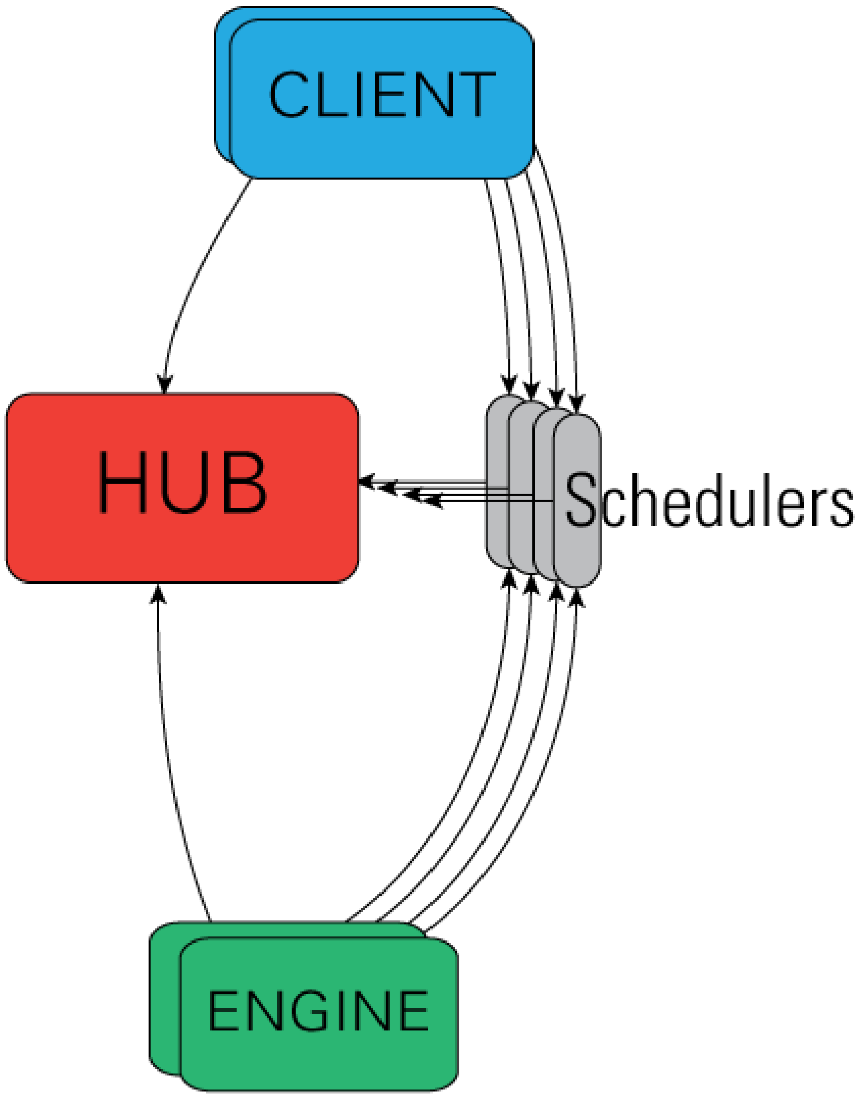
IPython engine

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


CLIENT
HUB
Schedulers
ENGINE

are implemented via the `View.apply()` method, after 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.

ipcontroller-client.json will look something like this, under default localhost circumstances:

```
{
  "url":"tcp:\/\/127.0.0.1:54424",
  "exec_key":"a361fe89-92fc-4762-9767-e2f0a05e3130",
  "ssh":"",
  "location":"10.19.1.135"
}
```

If, however, you are running the controller on a work node on a cluster, you will likely need to use ssh tunnels to connect clients from your laptop to it. You will also probably need to instruct the controller to listen for engines coming from other work nodes on the cluster. An example of ipcontroller-client.json, as created by:

```
$> ipcontroller --ip=* --ssh=login.mycluster.com
```

```
{
  "url":"tcp:\/\/*:54424",
  "exec_key":"a361fe89-92fc-4762-9767-e2f0a05e3130",
  "ssh":"login.mycluster.com",
  "location":"10.0.0.2"
}
```

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.

5.1.4 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('/path/to/my/ipcontroller-client.json', sshserver='me@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 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 = '*'
# c.HubFactory.location = '10.0.1.1'
```

Note: `--ip=*` instructs ZeroMQ to listen on all interfaces, but it does not contain the IP needed for engines / clients to know where the controller actually is. This can be specified with `--location=10.0.0.1`, the specific IP address of the controller, as seen from engines and/or clients. IPython tries to guess this value by default, but it will not always guess correctly. Check the `location` field in your connection files if you are having connection trouble.

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 `IPYTHONDIR/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 `IPYTHONDIR/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 `IPYTHONDIR/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, LSF), and Windows HPC Server.

In general, these are configured by the `IPClusterEngines.engine_set_launcher_class`, and `IPClusterStart.controller_launcher_class` configurables, which can be the fully specified object name (e.g. `'IPython.parallel.apps.launcher.LocalControllerLauncher'`), but if you are using IPython's builtin launchers, you can specify just the class name, or even just the prefix e.g:

```
c.IPClusterEngines.engine_launcher_class = 'SSH'
# equivalent to
c.IPClusterEngines.engine_launcher_class = 'SSHEngineSetLauncher'
# both of which expand to
c.IPClusterEngines.engine_launcher_class = 'IPython.parallel.apps.launcher.SSHEngineSetLaun
```

The shortest form being of particular use on the command line, where all you need to do to get an IPython cluster running with engines started with MPI is:

```
$> ipcluster start --engines=MPI
```

Assuming that the default MPI config is sufficient.

Note: shortcuts for builtin launcher names were added in 0.12, as was the `_class` suffix on the configurable names. If you use the old 0.11 names (e.g. `engine_set_launcher`), they will still work, but you will get a deprecation warning that the name has changed.

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 MPI launchers by adding the lines:

```
c.IPClusterEngines.engine_launcher_class = 'MPIEngineSetLauncher'
```

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_class = 'MPIControllerLauncher'
```

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_class = 'PBSControllerLauncher'
c.IPClusterEngines.engine_launcher_class = '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/mpiexec -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 you 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_class = 'SSHEngineSetLauncher'
# and if the Controller is also to be remote:
c.IPClusterStart.controller_launcher_class = '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.controller_args = ['--reuse', '--ip=*', '--profile-dir=/path/to/
```

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 and unsupported on Windows. Would require a working **ssh** on Windows. Also, we are using shell scripts to setup and execute commands on remote hosts.

Moving files with SSH

SSH launchers will try to move connection files, controlled by the `to_send` and `to_fetch` configurables. If your machines are on a shared filesystem, this step is unnecessary, and can be skipped by setting these to empty lists:

```
c.SSHLauncher.to_send = []
c.SSHLauncher.to_fetch = []
```

If our default guesses about paths don't work for you, or other files should be moved, you can manually specify these lists as tuples of (local_path, remote_path) for to_send, and (remote_path, local_path) for to_fetch. If you do specify these lists explicitly, IPython *will not* automatically send connection files, so you must include this yourself if they should still be sent/retrieved.

5.2.4 IPython on EC2 with StarCluster

The excellent StarCluster toolkit for managing Amazon EC2 clusters has a plugin which makes deploying IPython on EC2 quite simple. The starcluster plugin uses **ipcluster** with the SGE launchers to distribute engines across the EC2 cluster. See their ipcluster plugin documentation for more information.

5.2.5 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 `IPYTHONDIR/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`:

```
$ ipcontroller --ip=192.168.1.16
```

```
# in ipcontroller_config.py
HubFactory.ip = '192.168.1.16'
```

2. Copy `ipcontroller-engine.json` from `IPYTHONDIR/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 `IPYTHONDIR/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 (`IPYTHONDIR/profile_<name>/security` is the same on all of them), then things will just work!

SSH Tunnels

If your engines are not on the same LAN as the controller, or you are on a highly restricted network where your nodes cannot see each others ports, then you can use SSH tunnels to connect engines to the controller.

Note: This does not work in all cases. Manual tunnels may be an option, but are highly inconvenient. Support for manual tunnels will be improved.

You can instruct all engines to use ssh, by specifying the ssh server in `ipcontroller-engine.json`:

```
{
  "url":"tcp://192.168.1.123:56951",
  "exec_key":"26f4c040-587d-4a4e-b58b-030b96399584",
  "ssh":"user@example.com",
  "location":"192.168.1.123"
}
```

This will be specified if you give the `--enginessh=use@example.com` argument when starting **ipcontroller**.

Or you can specify an ssh server on the command-line when starting an engine:

```
$> ipengine --profile=foo --ssh=my.login.node
```

For example, if your system is totally restricted, then all connections will actually be loopback, and ssh tunnels will be used to connect engines to the controller:

```
[node1] $> ipcontroller --enginessh=node1
[node2] $> ipengine
[node3] $> ipcluster engines --n=4
```

Or if you want to start many engines on each node, the command *ipcluster engines –n=4* without any configuration is equivalent to running ipengine 4 times.

An example using ipcontroller/engine with ssh

No configuration files are necessary to use ipcontroller/engine in an SSH environment without a shared filesystem. You simply need to make sure that the controller is listening on an interface visible to the engines, and move the connection file from the controller to the engines.

1. start the controller, listening on an ip-address visible to the engine machines:

```
[controller.host] $ ipcontroller --ip=192.168.1.16

[IPControllerApp] Using existing profile dir: u'/Users/me/.ipython/profile_default'
[IPControllerApp] Hub listening on tcp://192.168.1.16:63320 for registration.
[IPControllerApp] Hub using DB backend: 'IPython.parallel.controller.dictdb.DictDB'
[IPControllerApp] hub::created hub
[IPControllerApp] writing connection info to /Users/me/.ipython/profile_default/securi
[IPControllerApp] writing connection info to /Users/me/.ipython/profile_default/securi
[IPControllerApp] task::using Python leastload Task scheduler
[IPControllerApp] Heartmonitor started
[IPControllerApp] Creating pid file: /Users/me/.ipython/profile_default/pid/ipcontroll
Scheduler started [leastload]
```

2. on each engine, fetch the connection file with scp:

```
[engine.host.n] $ scp controller.host:.ipython/profile_default/security/ipcontroller-e
```

Note: The log output of ipcontroller above shows you where the json files were written. They will be in `~/.ipython` (or `~/.config/ipython`) under `profile_default/security/ipcontroller-engine.json`

3. start the engines, using the connection file:

```
[engine.host.n] $ ipengine --file=./ipcontroller-engine.json
```

A couple of notes:

- You can avoid having to fetch the connection file every time by adding `--reuse` flag to ipcontroller, which instructs the controller to read the previous connection file for connection info, rather than generate a new one with randomized ports.
- In step 2, if you fetch the connection file directly into the security dir of a profile, then you need not specify its path directly, only the profile (assumes the path exists, otherwise you must create it first):

```
[engine.host.n] $ scp controller.host:.ipython/profile_default/security/ipcontroller-e
[engine.host.n] $ ipengine --profile=ssh
```

  Of course, if you fetch the file into the default profile, no arguments must be passed to ipengine at all.

- Note that ipengine *did not* specify the ip argument. In general, it is unlikely for any connection information to be specified at the command-line to ipengine, as all of this information should be contained in the connection file written by ipcontroller.

Make JSON files persistent

At fist glance it may seem that 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 is 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 `IPYTHONDIR/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
hostname = socket.gethostname()
# alternate choices for hostname include `socket.getfqdn()`
# or `socket.gethostname() + '.local'`

ex_ip = socket.gethostbyname_ex(hostname)[-1][-1]
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'

# You can use NoDB to disable the database altogether, in case you don't need
# to reuse tasks or results, and want to keep memory consumption under control.
c.HubFactory.db_class = 'IPython.parallel.controller.dictdb.NoDB'
```

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 = {}
```

But sometimes you are moving lots of data around quickly, and you don't need that information to be stored for later access, even by other Clients to this same session. For this case, we have a dummy database, which doesn't actually store anything. This lets the Hub stay small in memory, at the obvious expense of being able to access the information that would have been stored in the database (used for task resubmission, requesting results of tasks you didn't submit, etc.). To use this backend, simply pass `--nodb` to **`ipcontroller`** on the command-line, or specify the `NoDB` class in your `ipcontroller_config.py` as described above.

See also:

For more information on the database backends, see the *db backend reference*.

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 exection. 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 `DirectView` 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 `IPYTHONDIR/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
```

Calling a `@parallel` function *does not* correspond to map. It is used for splitting element-wise operations that operate on a sequence or array. For `map` behavior, parallel functions do have a map method.

call	pfunc(seq)	pfunc.map(seq)
# of tasks	# of engines (1 per engine)	# of engines (1 per engine)
# of remote calls	# of engines (1 per engine)	`len(seq)`
argument to remote	`seq[i:j]` (sub-sequence)	`seq[i]` (single element)

A quick example to illustrate the difference in arguments for the two modes:

```
In [16]: @dview.parallel(block=True)
   ....: def echo(x):
   ....:     return str(x)
   ....:

In [17]: echo(range(5))
Out[17]: ['[0, 1]', '[2]', '[3]', '[4]']

In [18]: echo.map(range(5))
Out[18]: ['0', '1', '2', '3', '4']
```

See also:

See 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 zeromq is done sending the message. 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.

dv.targets [int, list of ints] which targets this view is associated with.

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.

See also:

Docs on the *AsyncResult* object.

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()`) 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.
- The Keyword arguments, if provided overrides 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 [20]: 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.

See also:

See the documentation of the *Parallel Magics*.

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 [42]: dview.push(dict(c='speed'))
Out[42]: [None,None,None,None]
```

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

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

In [49]: ar.get()
Out[49]: [1.03234, 1.03234, 1.03234, 1.03234]
```

Dictionary interface

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

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

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

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know 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, pyzmq, or some other direct interconnect 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]

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. Failures of single Engines will be collected and raise a CompositeError, as demonstrated in the next section.

```
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
```

Note: `sync_imports()` does not allow `import foo as bar` syntax, because the assignment represented by the `as bar` part is not available to the import hook.

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 [78]: dview.block = True

In [79]: dview.execute("1/0")
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
```

```
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[2:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
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', block=True)
   ....: except parallel.error.CompositeError, e:
   ....:     e.raise_exception()
   ....:
   ....:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
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')
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[2:execute]:
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

In [82]: %debug
> /.../site-packages/IPython/parallel/client/asyncresult.py(125)get()
    124             else:
--> 125                 raise self._exception
    126         else:

# Here, self._exception is the CompositeError instance:

ipdb> e = self._exception
ipdb> e
CompositeError(4)

# we can tab-complete on e to see available methods:
ipdb> e.<TAB>
e.args               e.message            e.traceback
e.elist              e.msg
e.ename              e.print_traceback
e.engine_info        e.raise_exception
e.evalue             e.render_traceback

# We can then display the individual tracebacks, if we want:
ipdb> e.print_traceback(1)
[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero
```

Since you might have 100 engines, you probably don't want to see 100 tracebacks for a simple NameError because of a typo. For this reason, CompositeError truncates the list of exceptions it will print to `CompositeError.tb_limit` (default is five). You can change this limit to suit your needs with:

```
In [20]: from IPython.parallel import CompositeError
In [21]: CompositeError.tb_limit = 1
In [22]: %px a=b
[0:execute]:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
----> 1 a=b
NameError: name 'b' is not defined

... 3 more exceptions ...
```

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()
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
----> 1 1/0
ZeroDivisionError: integer division or modulo by zero

... 3 more exceptions ...
```

5.4 Parallel Magic Commands

We provide a few IPython magic commands that make it a bit more pleasant to execute Python commands on the engines interactively. These are mainly shortcuts to `DirectView.execute()` and `AsyncResult.display_outputs()` methods repsectively.

These magics will automatically become available when you create a Client:

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

The initially active View will have attributes `targets='all', block=True`, which is a blocking view of all engines, evaluated at request time (adding/removing engines will change where this view's tasks will run).

5.4.1 The Magics

%px

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

```
# import numpy here and everywhere
In [25]: with rc[:].sync_imports():
   ....:     import numpy
importing numpy on engine(s)

In [27]: %px a = numpy.random.rand(2,2)
Parallel execution on engines: [0, 1, 2, 3]

In [28]: %px numpy.linalg.eigvals(a)
Parallel execution on engines: [0, 1, 2, 3]
Out [0:68]: array([ 0.77120707, -0.19448286])
Out [1:68]: array([ 1.10815921,  0.05110369])
Out [2:68]: array([ 0.74625527, -0.37475081])
Out [3:68]: array([ 0.72931905,  0.07159743])
```

```
In [29]: %px print 'hi'
Parallel execution on engine(s): all
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

Since engines are IPython as well, you can even run magics remotely:

```
In [28]: %px %pylab inline
Parallel execution on engine(s): all
[stdout:0]
Populating the interactive namespace from numpy and matplotlib
[stdout:1]
Populating the interactive namespace from numpy and matplotlib
[stdout:2]
Populating the interactive namespace from numpy and matplotlib
[stdout:3]
Populating the interactive namespace from numpy and matplotlib
```

And once in pylab mode with the inline backend, you can make plots and they will be displayed in your frontend if it suports the inline figures (e.g. notebook or qtconsole):

```
In [40]: %px plot(rand(100))
Parallel execution on engine(s): all
<plot0>
<plot1>
<plot2>
<plot3>
Out[0:79]: [<matplotlib.lines.Line2D at 0x10a6286d0>]
Out[1:79]: [<matplotlib.lines.Line2D at 0x10b9476d0>]
Out[2:79]: [<matplotlib.lines.Line2D at 0x110652750>]
Out[3:79]: [<matplotlib.lines.Line2D at 0x10c6566d0>]
```

%%px Cell Magic

%%px can be used as a Cell Magic, which accepts some arguments for controlling the execution.

Targets and Blocking

%%px accepts `--targets` for controlling which engines on which to run, and `--[no]block` for specifying the blocking behavior of this cell, independent of the defaults for the View.

```
In [6]: %%px --targets ::2
   ...: print "I am even"
   ...:
Parallel execution on engine(s): [0, 2]
[stdout:0] I am even
[stdout:2] I am even
```

```
In [7]: %%px --targets 1
   ...: print "I am number 1"
   ...:
Parallel execution on engine(s): 1
I am number 1

In [8]: %%px
   ...: print "still 'all' by default"
   ...:
Parallel execution on engine(s): all
[stdout:0] still 'all' by default
[stdout:1] still 'all' by default
[stdout:2] still 'all' by default
[stdout:3] still 'all' by default

In [9]: %%px --noblock
   ...: import time
   ...: time.sleep(1)
   ...: time.time()
   ...:
Async parallel execution on engine(s): all
Out[9]: <AsyncResult: execute>

In [10]: %pxresult
Out[0:12]: 1339454561.069116
Out[1:10]: 1339454561.076752
Out[2:12]: 1339454561.072837
Out[3:10]: 1339454561.066665
```

See also:

%pxconfig accepts these same arguments for changing the *default* values of targets/blocking for the active View.

Output Display

%%px also accepts a `--group-outputs` argument, which adjusts how the outputs of multiple engines are presented.

See also:

`AsyncResult.display_outputs()` for the grouping options.

```
In [50]: %%px --block --group-outputs=engine
   ....: import numpy as np
   ....: A = np.random.random((2,2))
   ....: ev = numpy.linalg.eigvals(A)
   ....: print ev
   ....: ev.max()
   ....:
Parallel execution on engine(s): all
[stdout:0] [ 0.60640442  0.95919621]
Out [0:73]: 0.9591962130899806
```

```
[stdout:1] [ 0.38501813  1.29430871]
Out [1:73]: 1.2943087091452372
[stdout:2] [-0.85925141  0.9387692 ]
Out [2:73]: 0.93876920456230284
[stdout:3] [ 0.37998269  1.24218246]
Out [3:73]: 1.2421824618493817
```

%pxresult

If you are using %px in non-blocking mode, you won't get output. You can use %pxresult to display the outputs of the latest command, just as is done when %px is blocking:

```
In [39]: dv.block = False

In [40]: %px print 'hi'
Async parallel execution on engine(s): all

In [41]: %pxresult
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

%pxresult simply calls `AsyncResult.display_outputs()` on the most recent request. It accepts the same output-grouping arguments as %%px, so you can use it to view a result in different ways.

%autopx

The %autopx magic switches to a mode where everything you type is executed on the engines until you do %autopx again.

```
In [30]: dv.block=True

In [31]: %autopx
%autopx enabled

In [32]: max_evals = []

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)
   ....:

In [34]: print "Average max eigenvalue is: %f" % (sum(max_evals)/len(max_evals))
[stdout:0] Average max eigenvalue is: 10.193101
[stdout:1] Average max eigenvalue is: 10.064508
[stdout:2] Average max eigenvalue is: 10.055724
[stdout:3] Average max eigenvalue is: 10.086876
```

```
In [35]: %autopx
Auto Parallel Disabled
```

%pxconfig

The default targets and blocking behavior for the magics are governed by the `block` and `targets` attribute of the active View. If you have a handle for the view, you can set these attributes directly, but if you don't, you can change them with the %pxconfig magic:

```
In [3]: %pxconfig --block

In [5]: %px print 'hi'
Parallel execution on engine(s): all
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi

In [6]: %pxconfig --targets ::2

In [7]: %px print 'hi'
Parallel execution on engine(s): [0, 2]
[stdout:0] hi
[stdout:2] hi

In [8]: %pxconfig --noblock

In [9]: %px print 'are you there?'
Async parallel execution on engine(s): [0, 2]
Out[9]: <AsyncResult: execute>

In [10]: %pxresult
[stdout:0] are you there?
[stdout:2] are you there?
```

5.4.2 Multiple Active Views

The parallel magics are associated with a particular `DirectView` object. You can change the active view by calling the `activate()` method on any view.

```
In [11]: even = rc[::2]

In [12]: even.activate()

In [13]: %px print 'hi'
Async parallel execution on engine(s): [0, 2]
Out[13]: <AsyncResult: execute>

In [14]: even.block = True
In [15]: %px print 'hi'
Parallel execution on engine(s): [0, 2]
[stdout:0] hi
[stdout:2] hi
```

When activating a View, you can also specify a *suffix*, so that a whole different set of magics are associated with that view, without replacing the existing ones.

```
# restore the original DirecView to the base %px magics
In [16]: rc.activate()
Out[16]: <DirectView all>

In [17]: even.activate('_even')

In [18]: %px print 'hi all'
Parallel execution on engine(s): all
[stdout:0] hi all
[stdout:1] hi all
[stdout:2] hi all
[stdout:3] hi all

In [19]: %px_even print "We aren't odd!"
Parallel execution on engine(s): [0, 2]
[stdout:0] We aren't odd!
[stdout:2] We aren't odd!
```

This suffix is applied to the end of all magics, e.g. %autopx_even, %pxresult_even, etc.

For convenience, the `Client` has a `activate()` method as well, which creates a DirectView with block=True, activates it, and returns the new View.

The initial magics registered when you create a client are the result of a call to `rc.activate()` with default args.

5.4.3 Engines as Kernels

Engines are really the same object as the Kernels used elsewhere in IPython, with the minor exception that engines connect to a controller, while regular kernels bind their sockets, listening for connections from a QtConsole or other frontends.

Sometimes for debugging or inspection purposes, you would like a QtConsole connected to an engine for more direct interaction. You can do this by first instructing the Engine to *also* bind its kernel, to listen for connections:

```
In [50]: %px from IPython.parallel import bind_kernel; bind_kernel()
```

Then, if your engines are local, you can start a qtconsole right on the engine(s):

```
In [51]: %px %qtconsole
```

Careful with this one, because if your view is of 16 engines it will start 16 QtConsoles!

Or you can view just the connection info, and work out the right way to connect to the engines, depending on where they live and where you are:

```
In [51]: %px %connect_info
Parallel execution on engine(s): all
[stdout:0]
{
  "stdin_port": 60387,
  "ip": "127.0.0.1",
  "hb_port": 50835,
  "key": "eee2dd69-7dd3-4340-bf3e-7e2e22a62542",
  "shell_port": 55328,
  "iopub_port": 58264
}

Paste the above JSON into a file, and connect with:
    $> ipython <app> --existing <file>
or, if you are local, you can connect with just:
    $> ipython <app> --existing kernel-60125.json
or even just:
    $> ipython <app> --existing
if this is the most recent IPython session you have started.
[stdout:1]
{
  "stdin_port": 61869,
...
```

Note: `%qtconsole` will call `bind_kernel()` on an engine if it hasn't been done already, so you can often skip that first step.

5.5 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.

5.5.1 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.

5.5.2 Creating a `LoadBalancedView` 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*.

5.5.3 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,...]
```

5.5.4 Dependencies

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

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

Functional Dependencies

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

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

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

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

@require

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

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

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

@depend

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

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

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

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

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

dependents

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

Graph Dependencies

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In [17]: with lview.temp_flags(after=[ar,ar2]):
   ....:     ar3 = lview.apply(f3)

In [18]: with lview.temp_flags(follow=[ar], timeout=2.5)
   ....:     ar4 = lview.apply(f3)
```

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.

5.5.5 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.

5.5.6 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.

Greedy Assignment

Tasks can be assigned greedily as they are submitted. If their dependencies are met, they will be assigned to an engine right away, and multiple tasks can be assigned to an engine at a given time. This limit is set with the `TaskScheduler.hwm` (high water mark) configurable in your `ipcontroller_config.py` config file, with:

```
# the most common choices are:
c.TaskSheduler.hwm = 0 # (minimal latency, default in IPython < 0.13)
# or
c.TaskScheduler.hwm = 1 # (most-informed balancing, default in  0.13)
```

In IPython < 0.13, the default is 0, or no-limit. That is, there is no limit to the number of tasks that can be outstanding on a given engine. This greatly benefits the latency of execution, because network traffic can be hidden behind computation. However, this means that workload is assigned without knowledge of how long each task might take, and can result in poor load-balancing, particularly for submitting a collection of heterogeneous tasks all at once. You can limit this effect by setting hwm to a positive integer, 1 being maximum load-balancing (a task will never be waiting if there is an idle engine), and any larger number being a compromise between load-balancing and latency-hiding.

In practice, some users have been confused by having this optimization on by default, so the default value has been changed to 1 in IPython 0.13. This can be slower, but has more obvious behavior and won't result in assigning too many tasks to some engines in heterogeneous cases.

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 `DEALER` 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

5.5.7 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.

5.6 The AsyncResult object

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, but it also collects metadata on execution.

5.6.1 Beyond multiprocessing's AsyncResult

Note: The *`AsyncResult`* object provides a superset of the interface in multiprocessing.pool.AsyncResult. See the official Python documentation for more on the basics of this interface.

Our AsyncResult objects add a number of convenient features for working with parallel results, beyond what is provided by the original AsyncResult.

get_dict

First, is `AsyncResult.get_dict()`, which pulls results as a dictionary keyed by engine_id, rather than a flat list. This is useful for quickly coordinating or distributing information about all of the engines.

As an example, here is a quick call that gives every engine a dict showing the PID of every other engine:

```
In [10]: ar = rc[:].apply_async(os.getpid)
In [11]: pids = ar.get_dict()
In [12]: rc[:]['pid_map'] = pids
```

This trick is particularly useful when setting up inter-engine communication, as in IPython's `examples/parallel/interengine` examples.

5.6.2 Metadata

IPython.parallel tracks some metadata about the tasks, which is stored in the `Client.metadata` dict. The AsyncResult object gives you an interface for this information as well, including timestamps stdout/err, and engine IDs.

Timing

IPython tracks various timestamps as `datetime` objects, and the AsyncResult object has a few properties that turn these into useful times (in seconds as floats).

For use while the tasks are still pending:

- `ar.elapsed` is just the elapsed seconds since submission, for use before the AsyncResult is complete.
- `ar.progress` is the number of tasks that have completed. Fractional progress would be:

```
    1.0 * ar.progress / len(ar)
```

- `AsyncResult.wait_interactive()` will wait for the result to finish, but print out status updates on progress and elapsed time while it waits.

For use after the tasks are done:

- `ar.serial_time` is the sum of the computation time of all of the tasks done in parallel.
- `ar.wall_time` is the time between the first task submitted and last result received. This is the actual cost of computation, including IPython overhead.

Note: wall_time is only precise if the Client is waiting for results when the task finished, because the *received* timestamp is made when the result is unpacked by the Client, triggered by the `spin()` call. If you are doing work in the Client, and not waiting/spinning, then *received* might be artificially high.

An often interesting metric is the time it actually cost to do the work in parallel relative to the serial computation, and this can be given simply with

```
speedup = ar.serial_time / ar.wall_time
```

5.6.3 Map results are iterable!

When an AsyncResult object has multiple results (e.g. the `AsyncMapResult` object), you can actually iterate through results themselves, and act on them as they arrive:

```
from __future__ import print_function

import time

from IPython import parallel

# create client & view
rc = parallel.Client()
dv = rc[:]
v = rc.load_balanced_view()

# scatter 'id', so id=0,1,2 on engines 0,1,2
dv.scatter('id', rc.ids, flatten=True)
print("Engine IDs: ", dv['id'])

# create a Reference to `id`. This will be a different value on each engine
ref = parallel.Reference('id')
print("sleeping for `id` seconds on each engine")
tic = time.time()
ar = dv.apply(time.sleep, ref)
for i,r in enumerate(ar):
    print("%i: %.3f"%(i, time.time()-tic))

def sleep_here(t):
    import time
    time.sleep(t)
    return id,t

# one call per task
print("running with one call per task")
amr = v.map(sleep_here, [.01*t for t in range(100)])
tic = time.time()
for i,r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time()-tic))

print("running with four calls per task")
```

```
# with chunksize, we can have four calls per task
amr = v.map(sleep_here, [.01*t for t in range(100)], chunksize=4)
tic = time.time()
for i,r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time()-tic))

print("running with two calls per task, with unordered results")
# We can even iterate through faster results first, with ordered=False
amr = v.map(sleep_here, [.01*t for t in range(100,0,-1)], ordered=False, chunksize=2)
tic = time.time()
for i,r in enumerate(amr):
    print("slept %.2fs on engine %i: %.3f" % (r[1], r[0], time.time()-tic))
```

That is to say, if you treat an AsyncMapResult as if it were a list of your actual results, it should behave as you would expect, with the only difference being that you can start iterating through the results before they have even been computed.

This lets you do a dumb version of map/reduce with the builtin Python functions, and the only difference between doing this locally and doing it remotely in parallel is using the asynchronous view.map instead of the builtin map.

Here is a simple one-line RMS (root-mean-square) implemented with Python's builtin map/reduce.

```
In [38]: X = np.linspace(0,100)

In [39]: from math import sqrt

In [40]: add = lambda a,b: a+b

In [41]: sq = lambda x: x*x

In [42]: sqrt(reduce(add, map(sq, X)) / len(X))
Out[42]: 58.028845747399714

In [43]: sqrt(reduce(add, view.map(sq, X)) / len(X))
Out[43]: 58.028845747399714
```

To break that down:

1. `map(sq, X)` Compute the square of each element in the list (locally, or in parallel)
2. `reduce(add, sqX) / len(X)` compute the mean by summing over the list (or AsyncMapResult) and dividing by the size
3. take the square root of the resulting number

See also:

When AsyncResult or the AsyncMapResult don't provide what you need (for instance, handling individual results as they arrive, but with metadata), you can always just split the original result's `msg_ids` attribute, and handle them as you like.

For an example of this, see `examples/parallel/customresult.py`

5.7 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.

5.7.1 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.

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

```
$ ipcluster start -n 4 --engines=MPIEngineSetLauncher
```

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*.

5.7.3 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`:

```python
from mpi4py import MPI
import numpy as np

def psum(a):
    locsum = np.sum(a)
    rcvBuf = np.array(0.0,'d')
    MPI.COMM_WORLD.Allreduce([locsum, MPI.DOUBLE],
        [rcvBuf, MPI.DOUBLE],
        op=MPI.SUM)
    return rcvBuf
```

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 distributed array and sum up all its elements in a distributed manner using our `psum()` function:

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

In [2]: c = Client(profile='mpi')
In [3]: view = c[:]

In [4]: view.activate() # enable magics

# run the contents of the file on each engine:
In [5]: view.run('psum.py')

In [6]: view.scatter('a',np.arange(16,dtype='float'))

In [7]: view['a']
Out[7]: [array([ 0.,  1.,  2.,  3.]),
         array([ 4.,  5.,  6.,  7.]),
         array([  8.,   9.,  10.,  11.]),
         array([ 12.,  13.,  14.,  15.])]

In [7]: %px totalsum = psum(a)
Parallel execution on engines: [0,1,2,3]

In [8]: view['totalsum']
Out[8]: [120.0, 120.0, 120.0, 120.0]
```

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.

5.8 IPython's Task Database

5.8.1 Enabling a DB Backend

The IPython Hub can store all task requests and results in a database. Currently supported backends are: MongoDB, SQLite, and an in-memory DictDB.

This database behavior is optional due to its potential *Cost*, so you must enable one, either at the command-line:

```
$> ipcontroller --dictb # or --mongodb or --sqlitedb
```

or in your `ipcontroller_config.py`:

```
c.HubFactory.db_class = "DictDB"
c.HubFactory.db_class = "MongoDB"
c.HubFactory.db_class = "SQLiteDB"
```

5.8.2 Using the Task Database

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

MongoDB operators we emulate on all backends:

Operator	Python equivalent
'$in'	in
'$nin'	not in
'$eq'	==
'$ne'	!=
'$ge'	>
'$gte'	>=
'$le'	<
'$lte'	<=

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,...)

5.8.3 Example Queries

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

```
In [1]: incomplete = rc.db_query({'completed' : 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]: uuids = map(rc._engines.get, (3,4))

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

5.8.4 Cost

The advantage of the database backends is, of course, that large amounts of data can be stored that won't fit in memory. The basic DictDB 'backend' is actually to just store all of this information in a Python dictionary. This is very fast, but will run out of memory quickly if you move a lot of data around, or your cluster is to run for a long time.

Unfortunately, the DB backends (SQLite and MongoDB) right now are rather slow, and can still consume large amounts of resources, particularly if large tasks or results are being created at a high frequency.

For this reason, we have added `NoDB`,a dummy backend that doesn't actually store any information. When you use this database, nothing is stored, and any request for results will result in a KeyError. This obviously prevents later requests for results and task resubmission from functioning, but sometimes those nice features are not as useful as keeping Hub memory under control.

5.9 Security details of IPython

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

IPython's `IPython.kernel.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.

5.9.1 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.

5.9.2 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).

5.9.3 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.

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 signatures protect against unauthorized messages, but, as there is no encryption, provide exactly no protection of data privacy. It is possible, however, to use a custom serialization scheme (via Session.packer/unpacker traits) that does incorporate your own encryption scheme.

5.9.4 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.

5.9.5 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.

5.9.6 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.

5.10 Getting started with Windows HPC Server 2008

5.10.1 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 threads 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).

5.10.2 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.

5.10.3 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.org)

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:

```
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

Z:\>ipython
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.12.dev -- 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.

In [1]:
```

5.10.4 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. The result should look something like this:

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

Z:\>ipcluster start --profile=mycluster
```

```
[IPClusterStart] Using existing profile dir: u'\\\\blue\\domainusers$\\bgranger\\.ipython\\
[IPClusterStart] Starting ipcluster with [daemon=False]
[IPClusterStart] Creating pid file: \\blue\domainusers$\bgranger\.ipython\profile_myclustel
[IPClusterStart] Writing job description file: \\blue\domainusers$\bgranger\.ipython\profil
[IPClusterStart] Starting Win HPC Job: job submit /jobfile:\\blue\domainusers$\bgranger\.ip
[IPClusterStart] Starting 15 engines
[IPClusterStart] Writing job description file: \\blue\domainusers$\bgranger\.ipython\profil
[IPClusterStart] Starting Win HPC Job: job submit /jobfile:\\blue\domainusers$\bgranger\.ip
```

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 mycluster --parallel`
2. Edit configuration files in the directory `.ipython\cluster_mycluster`
3. Start the cluster using: `ipcluster 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 profile directory:

```
Z:\>ipython profile create mycluster --parallel
[ProfileCreate] Generating default config file: u'\\\\blue\\domainusers$\\bgranger\\.ipytho
[ProfileCreate] Generating default config file: u'\\\\blue\\domainusers$\\bgranger\\.ipytho
[ProfileCreate] Generating default config file: u'\\\\blue\\domainusers$\\bgranger\\.ipytho
[ProfileCreate] Generating default config file: u'\\\\blue\\domainusers$\\bgranger\\.ipytho
[ProfileCreate] Generating default config file: u'\\\\blue\\domainusers$\\bgranger\\.ipytho

Z:\>
```

Configuring a cluster profile

Next, you will need to configure the newly created cluster profile by editing the following configuration files in the cluster directory:

- `ipcluster_config.py`
- `ipcontroller_config.py`
- `ipengine_config.py`

When **`ipcluster`** is run, these configuration files are used to determine how the engines and controller will be started. In most cases, you will only have to set a few of the attributes in these files.

To configure **`ipcluster`** to use the Windows HPC job scheduler, you will need to edit the following attributes in the file `ipcluster_config.py`:

```
# Set these at the top of the file to tell ipcluster to use the
# Windows HPC job scheduler.
c.IPClusterStart.controller_launcher_class = 'WindowsHPCControllerLauncher'
c.IPClusterEngines.engine_launcher_class = '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 helpful to 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
```

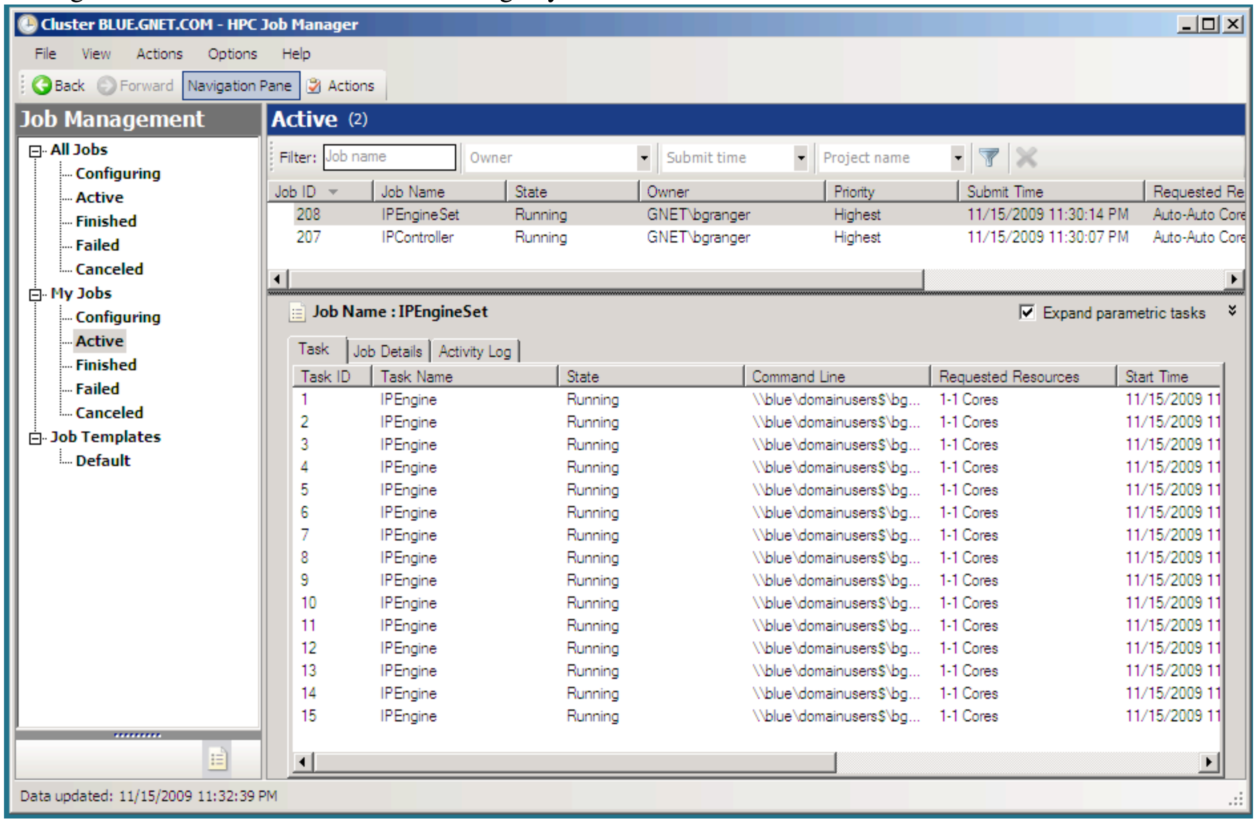
The `-n` option tells **`ipcluster`** how many engines to start (in this case 32). Stopping the cluster is as simple as typing Control-C.

Using the HPC Job Manager

føø 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.



Job ID	Job Name	State	Owner	Priority	Submit Time	Requested Re
208	IPEngineSet	Running	GNET\bgranger	Highest	11/15/2009 11:30:14 PM	Auto-Auto Core
207	IPController	Running	GNET\bgranger	Highest	11/15/2009 11:30:07 PM	Auto-Auto Core

Task ID	Task Name	State	Command Line	Requested Resources	Start Time
1	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
2	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
3	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
4	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
5	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
6	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
7	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
8	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
9	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
10	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
11	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
12	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
13	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
14	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11
15	IPEngine	Running	\\blue\domainusers$\bg...	1-1 Cores	11/15/2009 11

Data updated: 11/15/2009 11:32:39 PM

5.10.5 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 `DirectView` 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 `DirectView.map()` method:

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

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

In [3]: view = c[:]

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

```
In [5]: def f(x):
   ...:     return x**10

In [6]: view.map(f, range(15))  # f is applied in parallel
Out[6]:
[0,
 1,
 1024,
 59049,
 1048576,
 9765625,
 60466176,
 282475249,
 1073741824,
 3486784401L,
 10000000000L,
 25937424601L,
 61917364224L,
 137858491849L,
 289254654976L]
```

The `map()` method has the same signature as Python's builtin `map()` function, but runs the calculation in parallel. More involved examples of using `DirectView` are provided in the examples that follow.

5.11 Parallel examples

In this section we describe two more involved examples of using an IPython cluster to perform a parallel computation. We will be doing some plotting, so we start IPython with matplotlib integration by typing:

```
ipython --matplotlib
```

at the system command line. Or you can enable matplotlib integration at any point with:

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

5.11.1 150 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 `examples/parallel` 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
```

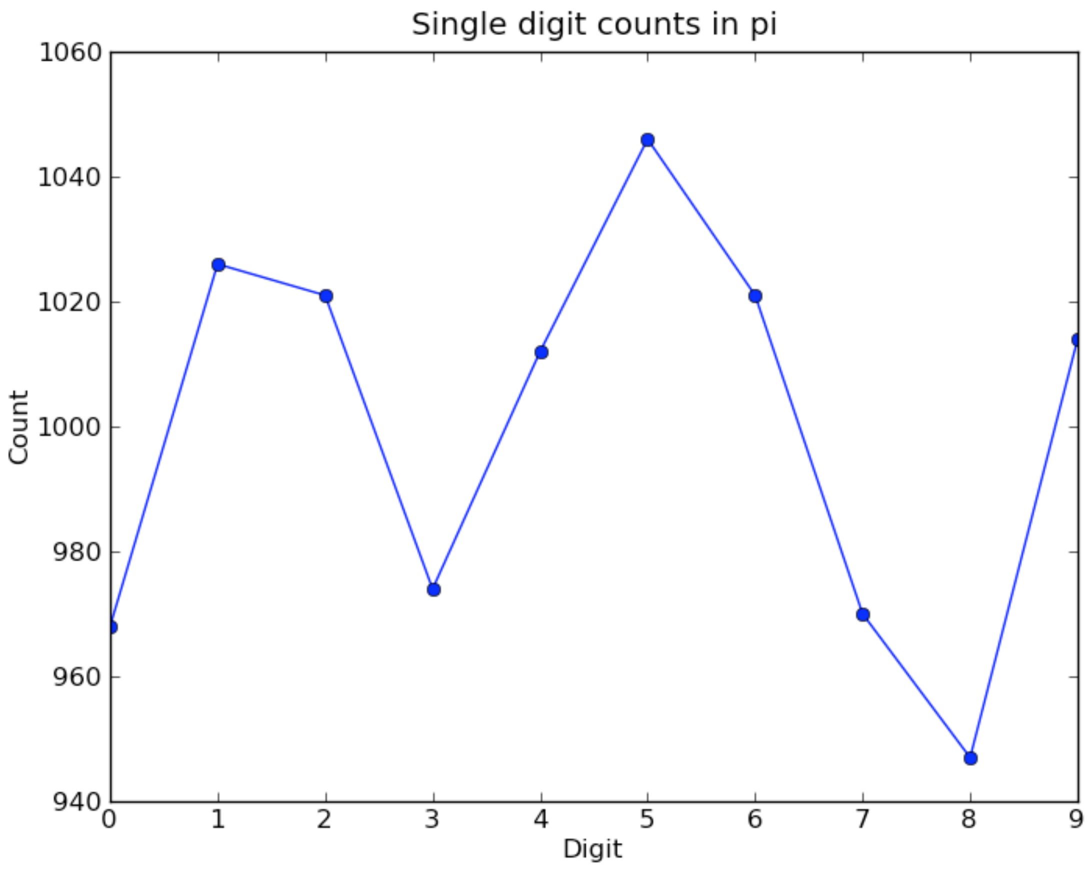
Serial calculation

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 [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 `examples/parallel/parallelpi.py`. This code can be run in parallel using IPython by following these steps:

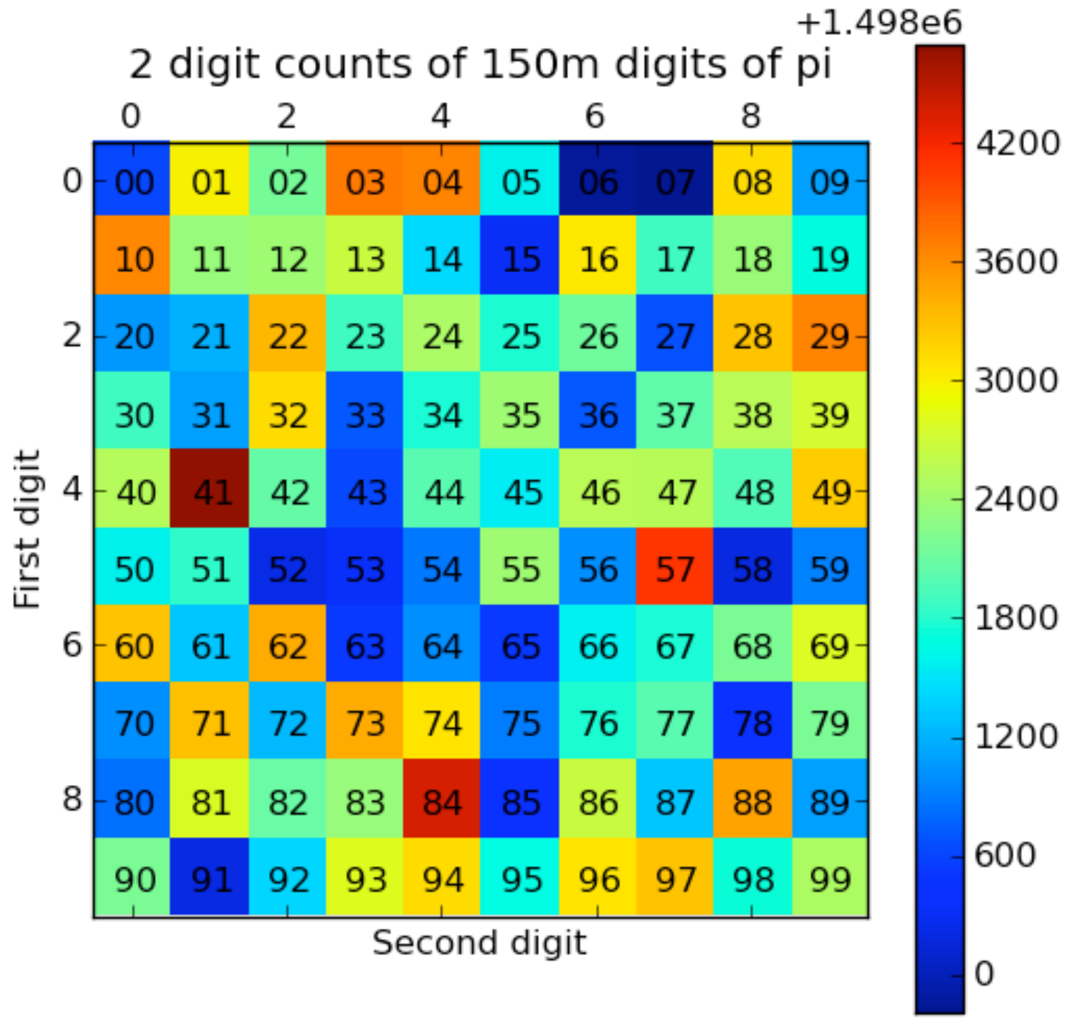
1. Use **`ipcluster`** to start 15 engines. We used 16 cores of an SGE linux cluster (1 controller + 15 engines).
2. With the file `parallelpi.py` in your current working directory, open up IPython, enable matplotlib, and type `run parallelpi.py`. This will download the pi files via ftp the first time you run it, if they are not present in the Engines' working directory.

When run on our 16 cores, we observe a speedup of 14.2x. This is slightly less than linear scaling (16x) 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 `parallelpi.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]: v = c[:]

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

In [4]: run pidigits.py

In [5]: filestring = 'pi200m.ascii.%(i)02dof20'

# Create the list of files to process.
In [6]: files = [filestring % {'i':i} for i in range(1,16)]

In [7]: files
Out[7]:
['pi200m.ascii.01of20',
 'pi200m.ascii.02of20',
 'pi200m.ascii.03of20',
 'pi200m.ascii.04of20',
 'pi200m.ascii.05of20',
 'pi200m.ascii.06of20',
 'pi200m.ascii.07of20',
 'pi200m.ascii.08of20',
 'pi200m.ascii.09of20',
 'pi200m.ascii.10of20',
 'pi200m.ascii.11of20',
 'pi200m.ascii.12of20',
 'pi200m.ascii.13of20',
 'pi200m.ascii.14of20',
 'pi200m.ascii.15of20']

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

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

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

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

In [12]: plt.title('2 digit counts of 150m digits of pi')
Out[12]: <matplotlib.text.Text object at 0x18d1f9b0>
```

The resulting plot generated by Matplotlib is shown below. The colors indicate which two digit sequences are more (red) or less (blue) likely to occur in the first 150 million digits of pi. We clearly see that the sequence "41" is most likely and that "06" and "07" are least likely. Further analysis would show that the relative size of the statistical fluctuations have decreased compared to the 10,000 digit calculation.



5.11.2 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 RHEL 5 and Sun GridEngine. IPython's built in support for SGE (and other batch systems) makes it easy to get started with IPython's parallel capabilities.

5.12 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 `examples/parallel/dagdeps.py`.

5.12.1 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.

5.12.2 A Sample DAG

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

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

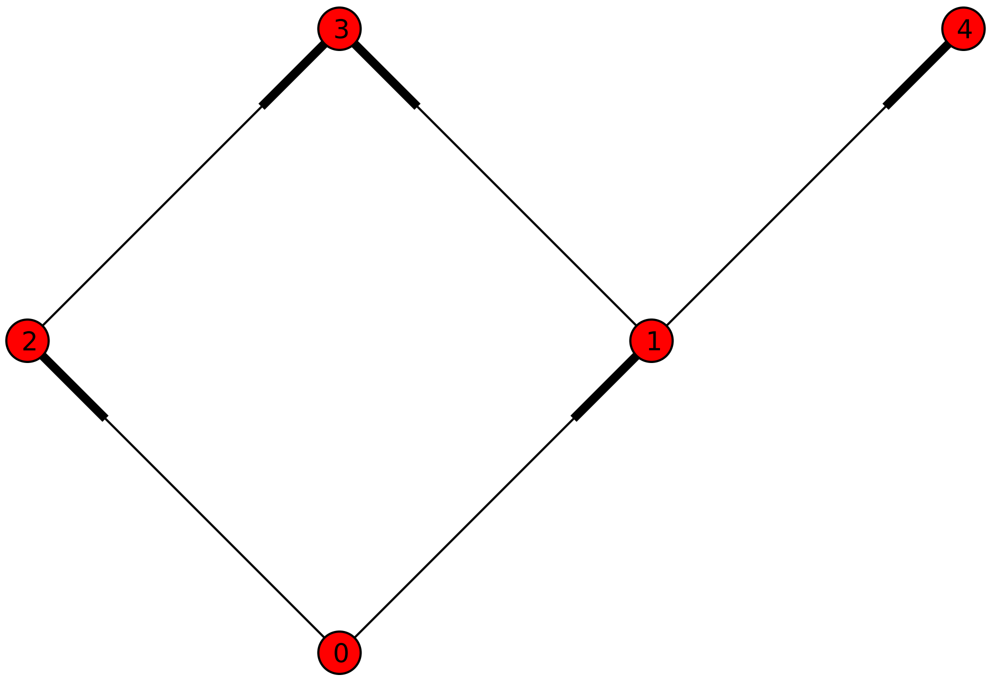
A possible sequence of events for this workflow:

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

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

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

The code to generate the simple DAG:


3
4
2
1
0

```
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 edg
    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
   ...:     with view.temp_flags(after=deps, block=False):
   ...:          results[node] = view.apply_with_flags(jobs[node])
```

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)
```

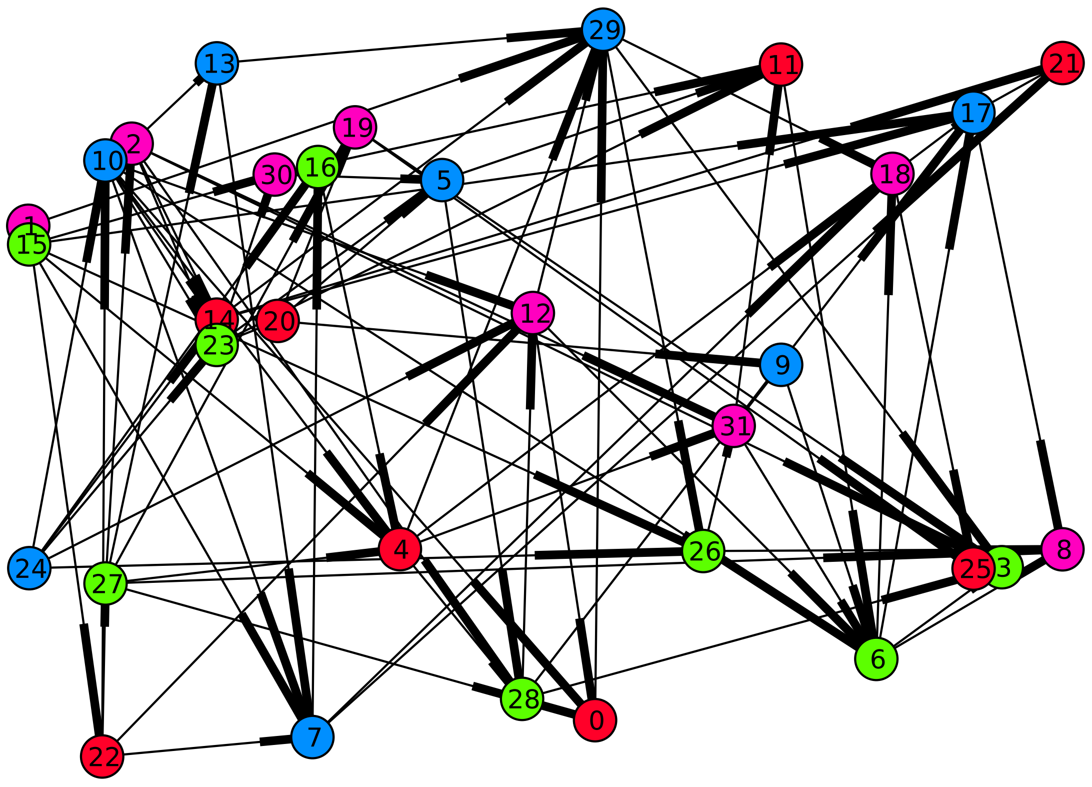


Fig. 5.1: Time started on x, runtime on y, and color-coded by engine-id (in this case there were four engines). Edges denote dependencies.

5.13 Details of Parallel Computing with IPython

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

5.13.1 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)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)<string> in <mod
<ipython-input-12-c3e7afeb3075> in setter(a)
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 0MQ to finish with a buffer.

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

In [6]: view.track=True
In [7]: ar = view.apply_async(lambda x: 2*x, A)

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

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

What is sendable?

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

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

Closures

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

An example of a function that uses a closure:

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

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

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

```
def g(b):
    # note that `b` is not referenced in inner's scope
    def inner():
        # this inner will *not* have a closure
        return a
    return inner
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 principle. 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')
```

5.13.2 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 fuction to be called remotely

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

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

flags for all views:

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

returns AsyncResult

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

list of results, matching *targets*

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

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

Run on all active engines

if list: Run on each specified engine

if int: Run on single engine

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

flags only in LoadBalancedViews:

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

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

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

execute and run

For executing strings of Python code, `DirectView` 's also provide an `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

5.13.3 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 know 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.

5.13.4 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.

5.13.5 Results

AsyncResults

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 AsyncResults when *block=False*.

The mp.pool.AsyncResult interface

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

class **`AsyncResult`**
The stdlib AsyncResult spec

`wait``(`[*timeout*]`)`
Wait until the result is available or until *timeout* seconds pass. This method always returns `None`.

`ready``()`
Return whether the call has completed.

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

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

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

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

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

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

See also:

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

Extended interface

Other extensions of the AsyncResult interface include convenience wrappers for `get()`. AsyncResults 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 AsyncResults.

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

5.13.6 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)

5.13.7 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.

5.13.8 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

5.13.9 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

5.13.10 Decorators and RemoteFunctions

Note: TODO: write this section

`@parallel()`

`@remote()`

`RemoteFunction`

`ParallelFunction`

5.13.11 Dependencies

Note: TODO: write this section

`@depend()`

`@require()`

`Dependency`

5.14 Transitioning from IPython.kernel to IPython.parallel

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

5.14.1 Processes

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

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

See also:

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

5.14.2 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()
```

5.14.3 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.

5.14.4 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.

5.14.5 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

CHAPTER 6

Configuration and customization

6.1 Overview of the IPython configuration system

This section describes the IPython configuration system.

The following discussion is for users who want to configure IPython to their liking. Developers who want to know how they can enable their objects to take advantage of the configuration system should consult the *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 things.

This `Configurable` is a subclass of `HasTraits` that knows how to configure itself. Class level traits with the metadata `config=True` become values that can be configured from the command line and configuration files.

Developers create `Configurable` subclasses that implement all of the logic in the application. Each of these subclasses has its own configuration information that controls how instances are created.

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 `IPYTHONDIR` environment variable and then default to a platform-specific default. Historical support for the `IPYTHON_DIR` environment variable will be removed in a future release.

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 `<IPYTHONDIR>/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.

Locating these files

From the command-line, you can quickly locate the IPYTHONDIR or a specific profile with:

```
$ ipython locate
/home/you/.ipython

$ ipython locate profile foo
/home/you/.ipython/profile_foo
```

These map to the utility functions: `IPython.utils.path.get_ipython_dir()` and `IPython.utils.path.locate_profile()` respectively.

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 `<IPYTHONDIR>/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.

Security Files

If you are using the notebook, qtconsole, or parallel code, IPython stores connection information in small JSON files in the active profile's security directory. This directory is made private, so only you can see the files inside. If you need to move connection files around to other computers, this is where they will be. If you want your code to be able to open security files by name, we have a convenience function `IPython.utils.path.get_security_file()`, which will return the absolute path to a security file from its filename and [optionally] profile name.

Startup Files

If you want some code to be run at the beginning of every IPython session with a particular profile, the easiest way is to add Python (`.py`) or IPython (`.ipy`) scripts to your `<profile>/startup` directory. Files in this directory will always be executed as soon as the IPython shell is constructed, and before any other code or scripts you have specified. If you have multiple files in the startup directory, they will be run in lexicographical order, so you can control the ordering by adding a '00-' prefix.

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.

Common Arguments

Since the strictness and verbosity of the KVLoader above are not ideal for everyday use, common arguments can be specified as *flags* or *aliases*.

Flags and Aliases are handled by `argparse` instead, allowing for more flexible parsing. In general, flags and aliases are prefixed by `--`, except for those that are single characters, in which case they can be specified with a single `-`, e.g.:

```
$ ipython -i -c "import numpy; x=numpy.linspace(0,1)" --profile testing --colors=lightbg
```

Aliases

For convenience, applications have a mapping of commonly used traits, so you don't have to specify the whole class name:

```
$ ipython --profile myprofile
# and
$ ipython --profile='myprofile'
# are equivalent to
$ ipython --BaseIPythonApplication.profile='myprofile'
```

Flags

Applications can also be passed **flags**. Flags are options that take no arguments. 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 --matploitlib
# is equivalent to
$ ipython --matplotlib auto
# or
$ ipython --no-banner
# is equivalent to
$ ipython --TerminalIPythonApp.display_banner=False
```

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

A level above configuration are IPython extensions, Python modules which modify the behaviour of the shell. They are referred to by an importable module name, and can be placed anywhere you'd normally import from, or in `$IPYTHONDIR/extensions/`.

6.2.1 Getting extensions

A few important extensions are *bundled with IPython*. Others can be found on the extensions index on the wiki, and installed with the `%install_ext` magic function.

6.2.2 Using extensions

To load an extension while IPython is running, use the `%load_ext` magic:

```
In [1]: %load_ext myextension
```

To load it each time IPython starts, list it in your configuration file:

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

6.2.3 Writing extensions

An IPython extension is an importable Python module that has a couple of special functions to load and unload it. Here is a template:

```
# myextension.py

def load_ipython_extension(ipython):
    # The `ipython` argument is the currently active `InteractiveShell`
    # instance, which can be used in any way. This allows you to register
    # new magics or aliases, for example.
```

```
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.

`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.

Useful `InteractiveShell` methods include `register_magic_function()`, `push()` (to add variables to the user namespace) and `drop_by_id()` (to remove variables on unloading).

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(ip.ipython_dir, 'extensions')`. This directory is added to `sys.path` automatically.

When your extension is ready for general use, please add it to the extensions index.

6.2.4 Extensions bundled with IPython

autoreload

IPython extension to reload modules before executing user code.

`autoreload` reloads modules automatically before entering the execution of code typed at the IPython prompt.

This makes for example the following workflow possible:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: from foo import some_function

In [4]: some_function()
Out[4]: 42

In [5]: # open foo.py in an editor and change some_function to return 43

In [6]: some_function()
Out[6]: 43
```

The module was reloaded without reloading it explicitly, and the object imported with `from foo import ...` was also updated.

Usage

The following magic commands are provided:

`%autoreload`

Reload all modules (except those excluded by `%aimport`) automatically now.

`%autoreload 0`

Disable automatic reloading.

`%autoreload 1`

Reload all modules imported with `%aimport` every time before executing the Python code typed.

`%autoreload 2`

Reload all modules (except those excluded by `%aimport`) every time before executing the Python code typed.

`%aimport`

List modules which are to be automatically imported or not to be imported.

`%aimport foo`

Import module ‘foo’ and mark it to be autoreloaded for `%autoreload 1`

`%aimport -foo`

Mark module ‘foo’ to not be autoreloaded.

Caveats

Reloading Python modules in a reliable way is in general difficult, and unexpected things may occur. `%autoreload` tries to work around common pitfalls by replacing function code objects and parts of classes previously in the module with new versions. This makes the following things to work:

- Functions and classes imported via ‘from xxx import foo’ are upgraded to new versions when ‘xxx’ is reloaded.
- Methods and properties of classes are upgraded on reload, so that calling ‘c.foo()’ on an object ‘c’ created before the reload causes the new code for ‘foo’ to be executed.

Some of the known remaining caveats are:

- Replacing code objects does not always succeed: changing a @property in a class to an ordinary method or a method to a member variable can cause problems (but in old objects only).
- Functions that are removed (eg. via monkey-patching) from a module before it is reloaded are not upgraded.
- C extension modules cannot be reloaded, and so cannot be autoreloaded.

cythonmagic

octavemagic

rmagic

storemagic

%store magic for lightweight persistence.

Stores variables, aliases and macros in IPython's database.

To automatically restore stored variables at startup, add this to your `ipython_config.py` file:

```
c.StoreMagic.autorestore = True
```

sympyprinting

A print function that pretty prints sympy Basic objects.

> **moduleauthor** Brian Granger

Usage

Once the extension is loaded, Sympy Basic objects are automatically pretty-printed.

As of SymPy 0.7.2, maintenance of this extension has moved to SymPy under sympy.interactive.ipythonprinting, any modifications to account for changes to SymPy should be submitted to SymPy rather than changed here. This module is maintained here for backwards compatablitiy with old SymPy versions.

6.3 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 `profile_default/ipython_config.py` in your *IPython directory*. By setting the attributes in this file, you can configure the application. To create the default config file, run this command:

```
$ ipython profile create
```

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.

6.3.1 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 printer 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. See *IPython extensions* for more details about extensions.

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.

6.3.2 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`.

6.3.3 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.TerminalIPythonApp.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.xmode = 'Context'

c.PromptManager.in_template  = 'In [\#]: '
c.PromptManager.in2_template = '   .\D.: '
c.PromptManager.out_template = 'Out[\#]: '
c.PromptManager.justify = True

c.PrefilterManager.multi_line_specials = True

c.AliasManager.user_aliases = [
 ('la', 'ls -al')
]
```

6.4 Integrating your objects with IPython

6.4.1 Tab completion

To change the attributes displayed by tab-completing your object, define a `__dir__(self)` method for it. For more details, see the documentation of the built-in dir() function.

6.4.2 Rich display

The notebook and the Qt console can display richer representations of objects. To use this, you can define any of a number of `_repr_*_()` methods. Note that these are surrounded by single, not double underscores.

Both the notebook and the Qt console can display `svg`, `png` and `jpeg` representations. The notebook can also display `html`, `javascript`, and `latex`. If the methods don't exist, or return `None`, it falls back to a standard `repr()`.

For example:

```
class Shout(object):
    def __init__(self, text):
        self.text = text

    def _repr_html_(self):
        return "<h1>" + self.text + "</h1>"
```

6.4.3 Custom exception tracebacks

Rarely, you might want to display a different traceback with an exception - IPython's own parallel computing framework does this to display errors from the engines. To do this, define a `_render_traceback_(self)` method which returns a list of strings, each containing one line of the traceback.

Please be conservative in using this feature; by replacing the default traceback you may hide important information from the user.

6.5 Editor configuration

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

- Editors (such as (X)Emacs, vim and 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 by setting `TerminalInteractiveShell.editor` in `ipython_config.py`.

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

Paul Ivanov's vim-ipython provides powerful IPython integration for vim.

6.5.3 (X)Emacs

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 with `matplotlib` integration and hardcoded light-background colors, you can use:

```
(setq py-python-command-args '("--matplotlib" "--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 Custom input transformation

IPython extends Python syntax to allow things like magic commands, and help with the `?` syntax. There are several ways to customise how the user's input is processed into Python code to be executed.

These hooks are mainly for other projects using IPython as the core of their interactive interface. Using them carelessly can easily break IPython!

6.6.1 String based transformations

When the user enters a line of code, it is first processed as a string. By the end of this stage, it must be valid Python syntax.

These transformers all subclass `IPython.core.inputtransformer.InputTransformer`, and are used by `IPython.core.inputsplitter.IPythonInputSplitter`.

These transformers act in three groups, stored separately as lists of instances in attributes of `IPythonInputSplitter`:

- `physical_line_transforms` act on the lines as the user enters them. For example, these strip Python prompts from examples pasted in.
- `logical_line_transforms` act on lines as connected by explicit line continuations, i.e. `\` at the end of physical lines. They are skipped inside multiline Python statements. This is the point where IPython recognises `%magic` commands, for instance.
- `python_line_transforms` act on blocks containing complete Python statements. Multi-line strings, lists and function calls are reassembled before being passed to these, but note that function and class *definitions* are still a series of separate statements. IPython does not use any of these by default.

An InteractiveShell instance actually has two `IPythonInputSplitter` instances, as the attributes `input_splitter`, to tell when a block of input is complete, and `input_transformer_manager`, to transform complete cells. If you add a transformer, you should make sure that it gets added to both.

Stateless transformations

The simplest kind of transformations work one line at a time. Write a function which takes a line and returns a line, and decorate it with `StatelessInputTransformer.wrap()`:

```
@StatelessInputTransformer.wrap
def my_special_commands(line):
    if line.startswith("¬"):
        return "specialcommand(" + repr(line) + ")"
    return line
```

The decorator returns a factory function which will produce instances of `StatelessInputTransformer` using your function.

Coroutine transformers

More advanced transformers can be written as coroutines. The coroutine will be sent each line in turn, followed by `None` to reset it. It can yield lines, or `None` if it is accumulating text to yield at a later point. When reset, it should give up any code it has accumulated.

This code in IPython strips a constant amount of leading indentation from each line in a cell:

```
@CoroutineInputTransformer.wrap
def leading_indent():
    """Remove leading indentation.

    If the first line starts with a spaces or tabs, the same whitespace will be
    removed from each following line until it is reset.
    """
    space_re = re.compile(r'^[ \t]+')
    line = ''
    while True:
        line = (yield line)

        if line is None:
            continue

        m = space_re.match(line)
        if m:
            space = m.group(0)
            while line is not None:
                if line.startswith(space):
                    line = line[len(space):]
                line = (yield line)
        else:
            # No leading spaces - wait for reset
            while line is not None:
                line = (yield line)

leading_indent.look_in_string = True
```

Token-based transformers

There is an experimental framework that takes care of tokenizing and untokenizing lines of code. Define a function that accepts a list of tokens, and returns an iterable of output tokens, and decorate it with `TokenInputTransformer.wrap()`. These should only be used in `python_line_transforms`.

6.6.2 AST transformations

After the code has been parsed as Python syntax, you can use Python's powerful *Abstract Syntax Tree* tools to modify it. Subclass `ast.NodeTransformer`, and add an instance to `shell.ast_transformers`.

This example wraps integer literals in an `Integer` class, which is useful for mathematical frameworks that want to handle e.g. `1/3` as a precise fraction:

```
class IntegerWrapper(ast.NodeTransformer):
    """Wraps all integers in a call to Integer()"""
    def visit_Num(self, node):
        if isinstance(node.n, int):
            return ast.Call(func=ast.Name(id='Integer', ctx=ast.Load()),
                            args=[node], keywords=[])
        return node
```

6.7 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.

6.7.1 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.

6.7.2 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 $IPYTHONDIR/ipythonrc and set the colors option to the desired value.

6.7.3 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.

6.7.4 Fine-tuning your prompt

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

```
\#
    the prompt/history count number. This escape is automatically
    wrapped in the coloring codes for the currently active color scheme.
\N
    the 'naked' prompt/history count number: this is just the number
    itself, without any coloring applied to it. This lets you produce
    numbered prompts with your own colors.
\D
    the prompt/history count, with the actual digits replaced by dots.
    Used mainly in continuation prompts (prompt_in2)
```

```
\w
    the current working directory
\W
    the basename of current working directory
\Xn
    where $n=0\ldots5.$ The current working directory, with $HOME
    replaced by ~, and filtered out to contain only $n$ path elements
\Yn
    Similar to \Xn, but with the $n+1$ element included if it is ~ (this
    is similar to the behavior of the %cn escapes in tcsh)
\u
    the username of the current user
\$
    if the effective UID is 0, a #, otherwise a $
\h
    the hostname up to the first '.'
\H
    the hostname
\n
    a newline
\r
    a carriage return
\v
    IPython version string
```

In addition to these, ANSI color escapes can be insterted into the prompts, as C_ColorName. The list of valid color names is: Black, Blue, Brown, Cyan, DarkGray, Green, LightBlue, LightCyan, LightGray, LightGreen, LightPurple, LightRed, NoColor, Normal, Purple, Red, White, Yellow.

Finally, IPython supports the evaluation of arbitrary expressions in your prompt string. The prompt strings are evaluated through the syntax of PEP 215, but basically you can use $x.y to expand the value of x.y, and for more complicated expressions you can use braces: ${foo()+x} will call function foo and add to it the value of x, before putting the result into your prompt. For example, using prompt_in1 '${commands.getoutput("uptime")}nIn [#]: ' will print the result of the uptime command on each prompt (assuming the commands module has been imported in your ipythonrc file).

Prompt examples

The following options in an ipythonrc file will give you IPython's default prompts:

```
prompt_in1 'In [\#]:'
prompt_in2 '   .\D.:'
prompt_out 'Out[\#]:'
```

which look like this:

```
In [1]: 1+2
Out[1]: 3

In [2]: for i in (1,2,3):
   ...:     print i,
   ...:
1 2 3
```

These will give you a very colorful prompt with path information:

```
#prompt_in1 '\C_Red\u\C_Blue[\C_Cyan\Y1\C_Blue]\C_LightGreen\#>'
prompt_in2 ' ..\D>'
prompt_out '<\#>'
```

which look like this:

```
fperez[~/ipython]1> 1+2
                <1> 3
fperez[~/ipython]2> for i in (1,2,3):
               ...>     print i,
               ...>
1 2 3
```

CHAPTER 7

IPython developer's guide

This are two categories of developer focused documentation:

1. Documentation for developers of *IPython itself*.
2. Documentation for developers of third party tools and libraries that use IPython.

This part of our documentation only contains information in the second category.

Developers interested in working on IPython itself should consult our developer information on the IPython GitHub wiki.

7.1 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:

7.1.1 Install git

Overview

Debian / Ubuntu	`sudo apt-get install git-core`
Fedora	`sudo yum install git-core`
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

7.1.2 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.

7.1.3 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?.

5. When you have finished, check you have committed all your changes:

```
    git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
    git format-patch -M -C master
```

You will now have several files named for the commits:

```
    0001-BF-added-tests-for-Funny-bug.patch
    0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the ipython mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the ipython repository on github - *Making your own copy (fork) of ipython*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/ipython.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

7.1.4 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.

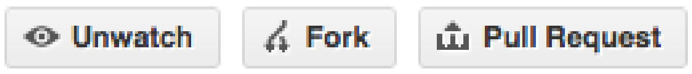
Set up and configure a github account

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

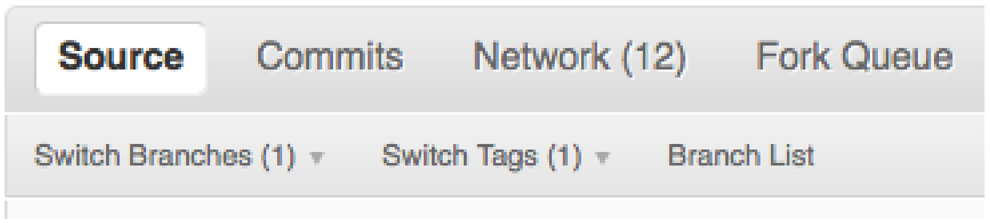
1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
    # 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")
```

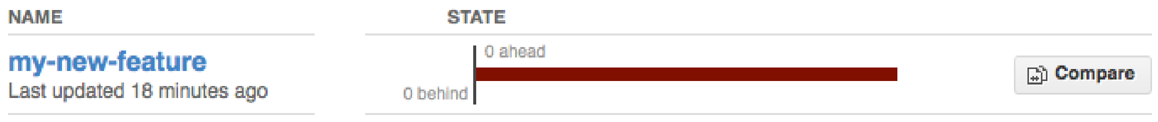
3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just 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`:



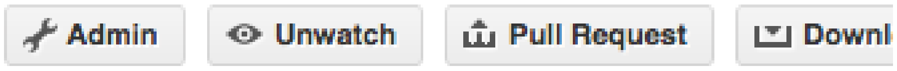
4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the ipython mailing list,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

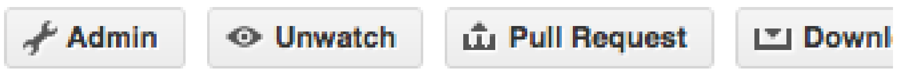
Several people sharing a single repository

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@githhub.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.

7.1.5 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
- Th git parable is an easy read explaining the concepts behind git.
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone

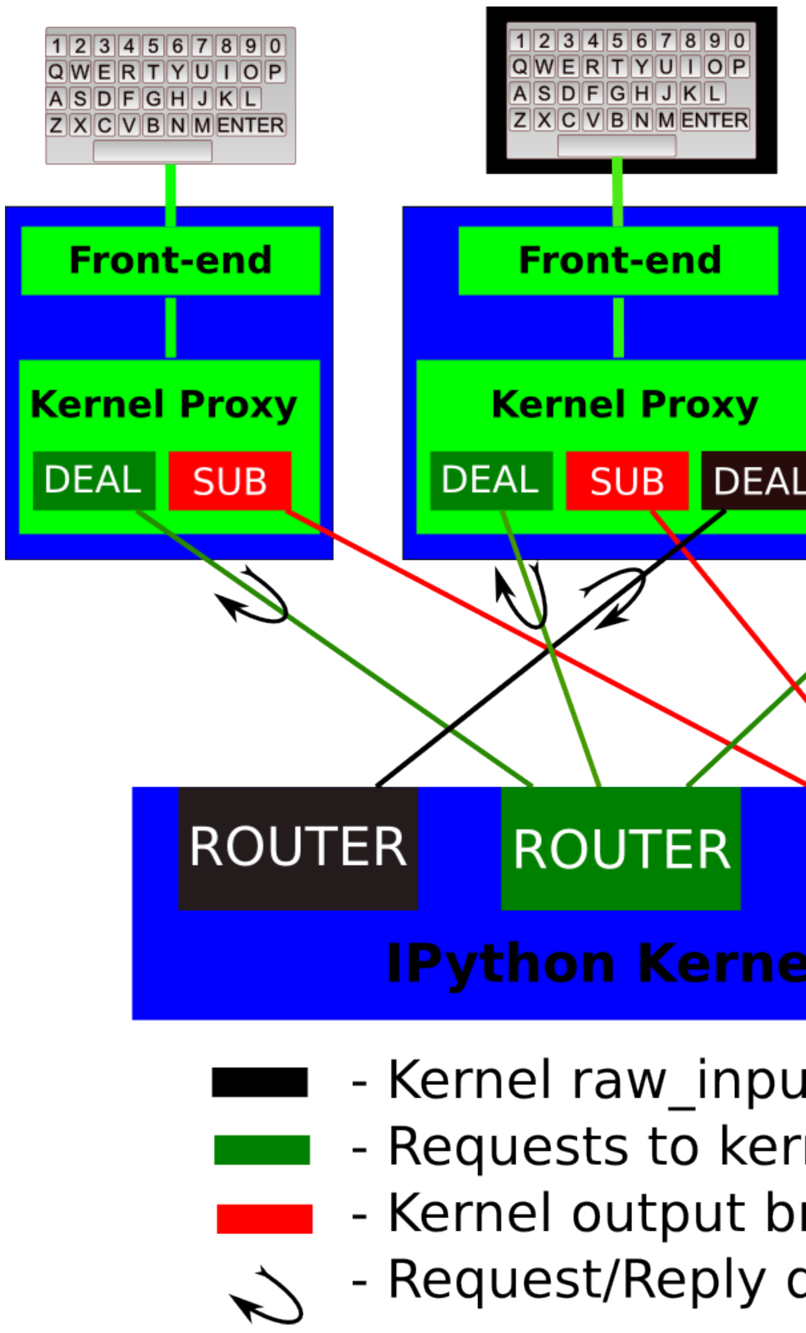
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

7.2 Messaging in IPython

7.2.1 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. stdin: this ROUTER socket is connected to all frontends, and it allows the kernel to request input from the active frontend when `raw_input()` is called. The frontend that executed the code has a DEALER socket that 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. Shell: this single ROUTER 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. IOPub: 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 shell socket and its own requests on the stdin 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 shell 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.

7.2.2 General Message Format

A message is defined by the following four-dictionary 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,
                # All recognized message type strings are listed below.
                'msg_type' : str,
     },

  # In a chain of messages, the header from the parent is copied so that
  # clients can track where messages come from.
  'parent_header' : dict,
  # Any metadata associated with the message.
  'metadata' : dict,

  # The actual content of the message must be a dict, whose structure
  # depends on the message type.
  'content' : dict,
}
```

7.2.3 The Wire Protocol

This message format exists at a high level, but does not describe the actual *implementation* at the wire level in zeromq. The canonical implementation of the message spec is our `Session` class.

Note: This section should only be relevant to non-Python consumers of the protocol. Python consumers should simply import and use IPython's own implementation of the wire protocol in the `IPython.kernel.zmq.session.Session` object.

Every message is serialized to a sequence of at least six blobs of bytes:

```
[
  b'u-u-i-d',          # zmq identity(ies)
  b'<IDS|MSG>',        # delimiter
  b'baddad42',         # HMAC signature
  b'{header}',         # serialized header dict
  b'{parent_header}',  # serialized parent header dict
  b'{metadata}',       # serialized metadata dict
  b'{content},         # serialized content dict
  b'blob',             # extra raw data buffer(s)
  ...
]
```

The front of the message is the ZeroMQ routing prefix, which can be zero or more socket identities. This is every piece of the message prior to the delimiter key `<IDS|MSG>`. In the case of IOPub, there should be just one prefix component, which is the topic for IOPub subscribers, e.g. `pyout`, `display_data`.

Note: In most cases, the IOPub topics are irrelevant and completely ignored, because frontends just subscribe to all topics. The convention used in the IPython kernel is to use the msg_type as the topic, and possibly extra information about the message, e.g. `pyout` or `stream.stdout`

After the delimiter is the HMAC signature of the message, used for authentication. If authentication is disabled, this should be an empty string. By default, the hashing function used for computing these signatures is sha256.

Note: To disable authentication and signature checking, set the *key* field of a connection file to an empty string.

The signature is the HMAC hex digest of the concatenation of:

- A shared key (typically the `key` field of a connection file)

- The serialized header dict
- The serialized parent header dict
- The serialized metadata dict
- The serialized content dict

In Python, this is implemented via:

```
# once:
digester = HMAC(key, digestmod=hashlib.sha256)

# for each message
d = digester.copy()
for serialized_dict in (header, parent, metadata, content):
    d.update(serialized_dict)
signature = d.hexdigest()
```

After the signature is the actual message, always in four frames of bytes. The four dictionaries that compose a message are serialized separately, in the order of header, parent header, metadata, and content. These can be serialized by any function that turns a dict into bytes. The default and most common serialization is JSON, but msgpack and pickle are common alternatives.

After the serialized dicts are zero to many raw data buffers, which can be used by message types that support binary data (mainly apply and data_pub).

7.2.4 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.

In addition, the Python implementation of the message specification extends messages upon deserialization to the following form for convenience:

```
{
  'header' : dict,
  # The msg's unique identifier and type are always stored in the header,
  # but the Python implementation copies them to the top level.
  'msg_id' : uuid,
  'msg_type' : str,
  'parent_header' : dict,
  'content' : dict,
  'metadata' : dict,
}
```

All messages sent to or received by any IPython process should have this extended structure.

7.2.5 Messages on the shell ROUTER/DEALER sockets

Execute

This message type is used by frontends to ask the kernel to execute code on behalf of the user, in a namespace reserved to the user's variables (and thus separate from the kernel's own internal code and variables).

Message type: `execute_request`:

```
content = {
    # Source code to be executed by the kernel, one or more lines.
'code' : str,

# A boolean flag which, if True, signals the kernel to execute
# this code as quietly as possible.  This means that the kernel
# will compile the code with 'exec' instead of 'single' (so
# sys.displayhook will not fire), forces store_history to be False,
# and will *not*:
#   - broadcast exceptions on the PUB socket
#   - do any logging
#
# The default is False.
'silent' : bool,

# A boolean flag which, if True, signals the kernel to populate history
# The default is True if silent is False.  If silent is True, store_history
# is forced to be False.
'store_history' : bool,

# A list of variable names from the user's namespace to be retrieved.
# What returns is a rich representation of each variable (dict keyed by name).
# See the display_data content for the structure of the representation data.
'user_variables' : list,

# Similarly, a dict mapping names to expressions to be evaluated in the
# user's dict.
'user_expressions' : dict,

# Some frontends (e.g. the Notebook) do not support stdin requests. If
# raw_input is called from code executed from such a frontend, a
# StdinNotImplementedError will be raised.
'allow_stdin' : True,

}
```

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 `store_history=True`. 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 request that
  # stores history.  This will typically be used by clients to display
  # prompt numbers to the user.  If the request did not store history, 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:

```
{
  # 'payload' will be a list of payload dicts.
  # Each 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' : list(dict),

  # Results for the user_variables and user_expressions.
  'user_variables' : dict,
  'user_expressions' : dict,
}
```

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 has an API for this in the PayloadManager:

```
ip.payload_manager.write_payload(payload_dict)
```

which appends a dictionary to the list of payloads.

The payload API is not yet stabilized, and should probably not be supported by non-Python kernels at this time. In such cases, the payload list should always be empty.

When status is 'error', the following extra fields are present:

```
{
  'ename' : str,    # Exception name, as a string
  'evalue' : 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.

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
    'oname' : 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'
    # this may be an empty string if the frontend does not do any lexing,
    # in which case the kernel must figure out the completion
    # based on 'line' and 'cursor_pos'.
    '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 or null/None,

    # 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,

# the substring of the matched text
# this is typically the common prefix of the matches,
# and the text that is already in the block that would be replaced by the full completion.
# This would be 'a.is' in the above example.
'matched_text' : str,

# status should be 'ok' unless an exception was raised during the request,
# in which case it should be 'error', along with the usual error message content
# in other messages.
'status' : 'ok'
}
```

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' or 'search', 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,

  # If hist_access_type is 'search' and unique is true, do not
  # include duplicated history.  Default is false.
  'unique' : bool,

}
```

New in version 4.0: The key `unique` for `history_request`.

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 shell channel) to connect to a kernel.

Message type: `connect_request`:

```
content = {
}
```

Message type: `connect_reply`:

```
content = {
    'shell_port' : int,    # The port the shell ROUTER socket is listening on.
    'iopub_port' : int,    # The port the PUB socket is listening on.
    'stdin_port' : int,    # The port the stdin ROUTER socket is listening on.
    'hb_port' : int,       # The port the heartbeat socket is listening on.
}
```

Kernel info

If a client needs to know information about the kernel, it can make a request of the kernel's information. This message can be used to fetch core information of the kernel, including language (e.g., Python), language version number and IPython version number, and the IPython message spec version number.

Message type: `kernel_info_request`:

```
content = {
}
```

Message type: `kernel_info_reply`:

```
content = {
    # Version of messaging protocol (mandatory).
    # The first integer indicates major version.  It is incremented when
    # there is any backward incompatible change.
    # The second integer indicates minor version.  It is incremented when
    # there is any backward compatible change.
    'protocol_version': [int, int],

    # IPython version number (optional).
    # Non-python kernel backend may not have this version number.
    # The last component is an extra field, which may be 'dev' or
    # 'rc1' in development version.  It is an empty string for
    # released version.
    'ipython_version': [int, int, int, str],

    # Language version number (mandatory).
    # It is Python version number (e.g., [2, 7, 3]) for the kernel
    # included in IPython.
    'language_version': [int, ...],

    # Programming language in which kernel is implemented (mandatory).
    # Kernel included in IPython returns 'python'.
    'language': str,
}
```

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.

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.

7.2.6 Messages on the PUB/SUB socket

Streams (stdout, stderr, etc)

Message type: `stream`:

```
content = {
    # The name of the stream is one of 'stdout', 'stderr'
    'name' : str,

    # The data is an arbitrary string to be written to that stream
    'data' : str,
}
```

Display Data

This type of message is used to bring back data that should be diplayed (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 secion.

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.
    'data' : dict,

    # Any metadata that describes the data
    'metadata' : dict
}
```

The `metadata` contains any metadata that describes the output. Global keys are assumed to apply to the output as a whole. The `metadata` dict can also contain mime-type keys, which will be sub-dictionaries, which are interpreted as applying only to output of that type. Third parties should put any data they write into a single dict with a reasonably unique name to avoid conflicts.

The only metadata keys currently defined in IPython are the width and height of images:

```
'metadata' : {
  'image/png' : {
    'width': 640,
    'height': 480
  }
}
```

Raw Data Publication

`display_data` lets you publish *representations* of data, such as images and html. This `data_pub` message lets you publish *actual raw data*, sent via message buffers.

data_pub messages are constructed via the `IPython.lib.datapub.publish_data()` function:

```
from IPython.kernel.zmq.datapub import publish_data
ns = dict(x=my_array)
publish_data(ns)
```

Message type: `data_pub`:

```
content = {
    # the keys of the data dict, after it has been unserialized
    keys = ['a', 'b']
}
# the namespace dict will be serialized in the message buffers,
# which will have a length of at least one
buffers = ['pdict', ...]
```

The interpretation of a sequence of data_pub messages for a given parent request should be to update a single namespace with subsequent results.

Note: No frontends directly handle data_pub messages at this time. It is currently only used by the client/engines in `IPython.parallel`, where engines may publish *data* to the Client, of which the Client can then publish *representations* via `display_data` to various frontends.

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

    # The counter for this execution is also provided so that clients can
    # display it, since IPython automatically creates variables called _iN
    # (for input prompt In[N]).
    'execution_count' : int
}
```

Python outputs

When Python produces output from code that has been compiled in with the 'single' flag to `compile()`, any expression that produces a value (such as `1+1`) is passed to `sys.displayhook`, which is a callable that can do with this value whatever it wants. The default behavior of `sys.displayhook` in the Python interactive prompt is to print to `sys.stdout` the `repr()` of the value as long as it is not `None` (which isn't printed at all). In our case, the kernel instantiates as `sys.displayhook` an object which has similar behavior, but which instead of printing to stdout, broadcasts these values as `pyout` messages for clients to display appropriately.

IPython's displayhook can handle multiple simultaneous formats depending on its configuration. The default pretty-printed repr text is always given with the `data` entry in this message. Any other formats are provided in the `extra_formats` list. Frontends are free to display any or all of these according to its capabilities. `extra_formats` list contains 3-tuples of an ID string, a type string, and the data. The ID is unique to the formatter implementation that created the data. Frontends will typically ignore the ID unless 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,

    # data and metadata are identical to a display_data message.
    # the object being displayed is that passed to the display hook,
    # i.e. the *result* of the execution.
    'data' : dict,
    'metadata' : 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.
    # The kernel will publish state 'starting' exactly once at process startup.
    execution_state : ('busy', 'idle', 'starting')
}
```

7.2.7 Messages on the stdin ROUTER/DEALER sockets

This is a socket where the request/reply pattern 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. The request should be made to the frontend that made the execution request that prompted `raw_input` to be called. For now we will keep these messages as simple as possible, since they 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.

7.2.8 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 DEALER 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 DEALER 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.

7.2.9 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.

7.3 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.

7.3.1 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

7.3.2 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 (`ROUTER`)

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 `DEALER` 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 = {
    'uuid'   : 'abcd-1234-...', # the zmq.IDENTITY of the engine's sockets
}
```

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
}
```

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'
}
```

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
    'uuid' : 'engine_id' # the IDENT for the engine's sockets
}
```

Message type : `unregistration_notification`:

```
content = {
    'id' : 0 # engine ID that has been unregistered
    'uuid' : 'engine_id' # the IDENT for the engine's sockets
}
```

Client Queries (ROUTER)

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

`queue_request()` requests can specify multiple engines to query via the *targets* element. A verbose flag can be passed, to determine whether the result should be the list of *msg_ids* in the queue or simply the length of each list.

Message type: `queue_request`:

```
content = {
    'verbose' : True, # whether return should be lists themselves or just lens
    'targets' : [0,3,1] # list of ints
}
```

The content of a reply to a `queue_request()` request is a dict, keyed by the engine IDs. Note that they will be the string representation of the integer keys, since JSON cannot handle number keys. The three keys of each dict are:

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

Message type: `queue_reply`:

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

Clients can request individual results directly from the hub. This is primarily for gathering results of executions not submitted by the requesting client, as the client will have all its own results already. Requests are made by msg_id, and can contain one or more msg_id. An additional boolean key 'statusonly' can be used to not request the results, but simply poll the status of the jobs.

Message type: `result_request`:

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

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

Message type: `result_reply`:

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

For memory management purposes, Clients can also instruct the hub to forget the results of messages. This can be done by message ID or engine ID. Individual messages are dropped by msg_id, and all messages completed on an engine are dropped by engine ID. This may no longer be necessary with the mongodb-based message logging backend.

If the msg_ids element is the string `'all'` instead of a list, then all completed results are forgotten.

Message type: `purge_request`:

```
content = {
    'msg_ids' : ['id1', 'id2',...], # list of msg_ids or 'all'
    'engine_ids' : [0,2,4] # list of engine IDs
}
```

The reply to a purge request is simply the status 'ok' if the request succeeded, or an explanation of why it failed, such as requesting the purge of a nonexistent or pending message.

Message type: `purge_reply`:

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

7.3.3 Schedulers

There are three basic schedulers:

- Task Scheduler
- MUX Scheduler
- Control Scheduler

The MUX and Control schedulers are simple MonitoredQueue ØMQ devices, with `ROUTER` 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 `ROUTER`, and the engine-facing socket is `DEALER`. The result of this is that client-submitted messages are load-balanced via the `DEALER` socket, but the engine's replies to each message go to the requesting client.

Raw `DEALER` 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 ROUTER 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()

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.kernel.zmq.serialize` 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`:

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

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

Message type: `apply_reply`:

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

```
buffers = ['...'] # either 1 or 2 in length
                  # a serialization of the return value of f(*args,**kwargs)
                  # only populated if status is 'ok'
```

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

7.3.4 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 by 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
}
```

7.3.5 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 4 parts: the header, the parent header, the metadata 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.

7.4 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.

7.4.1 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.

The Controller consists of 1-5 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.

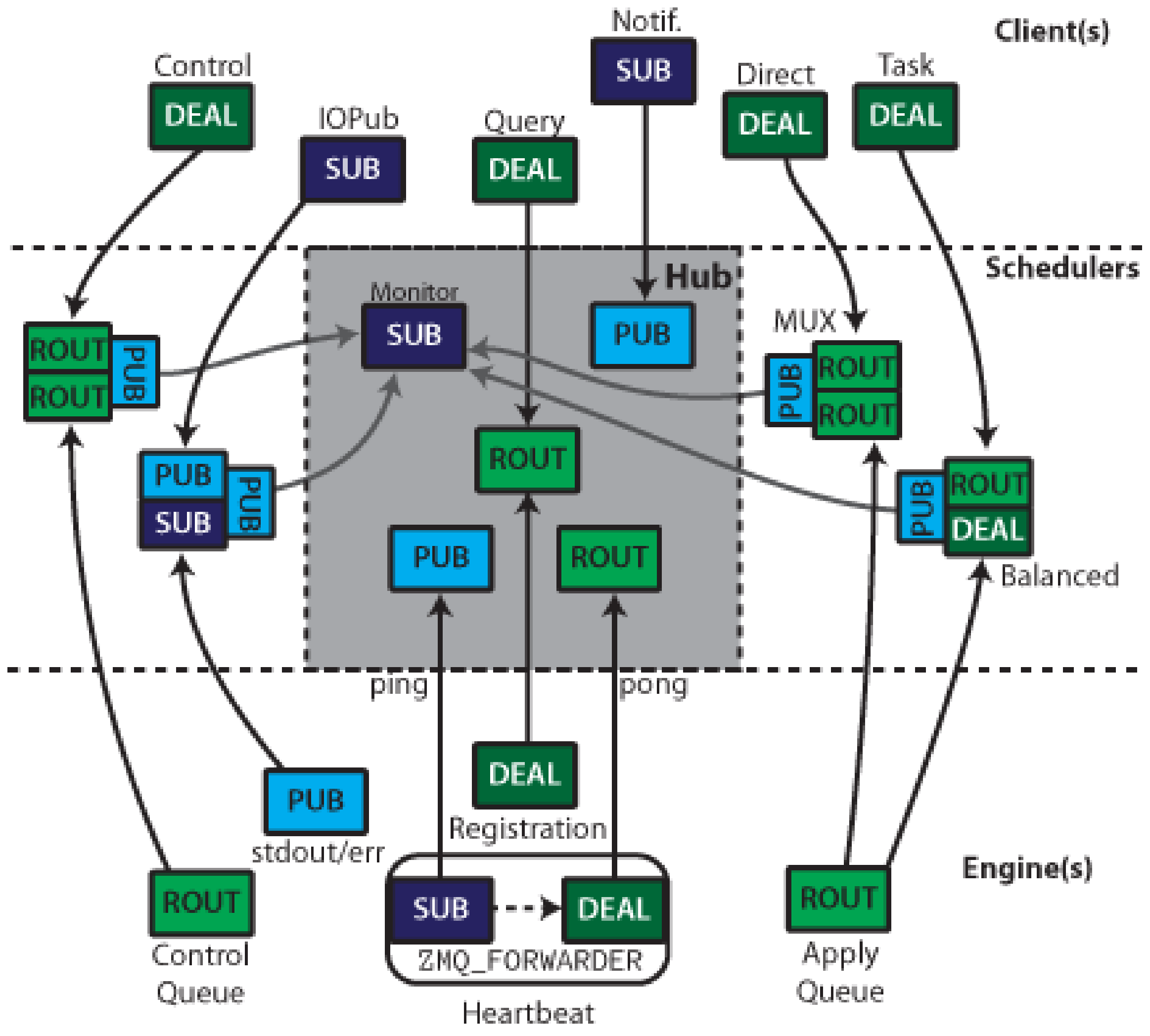


Fig. 7.1: All the connections involved in connecting one client to one engine.

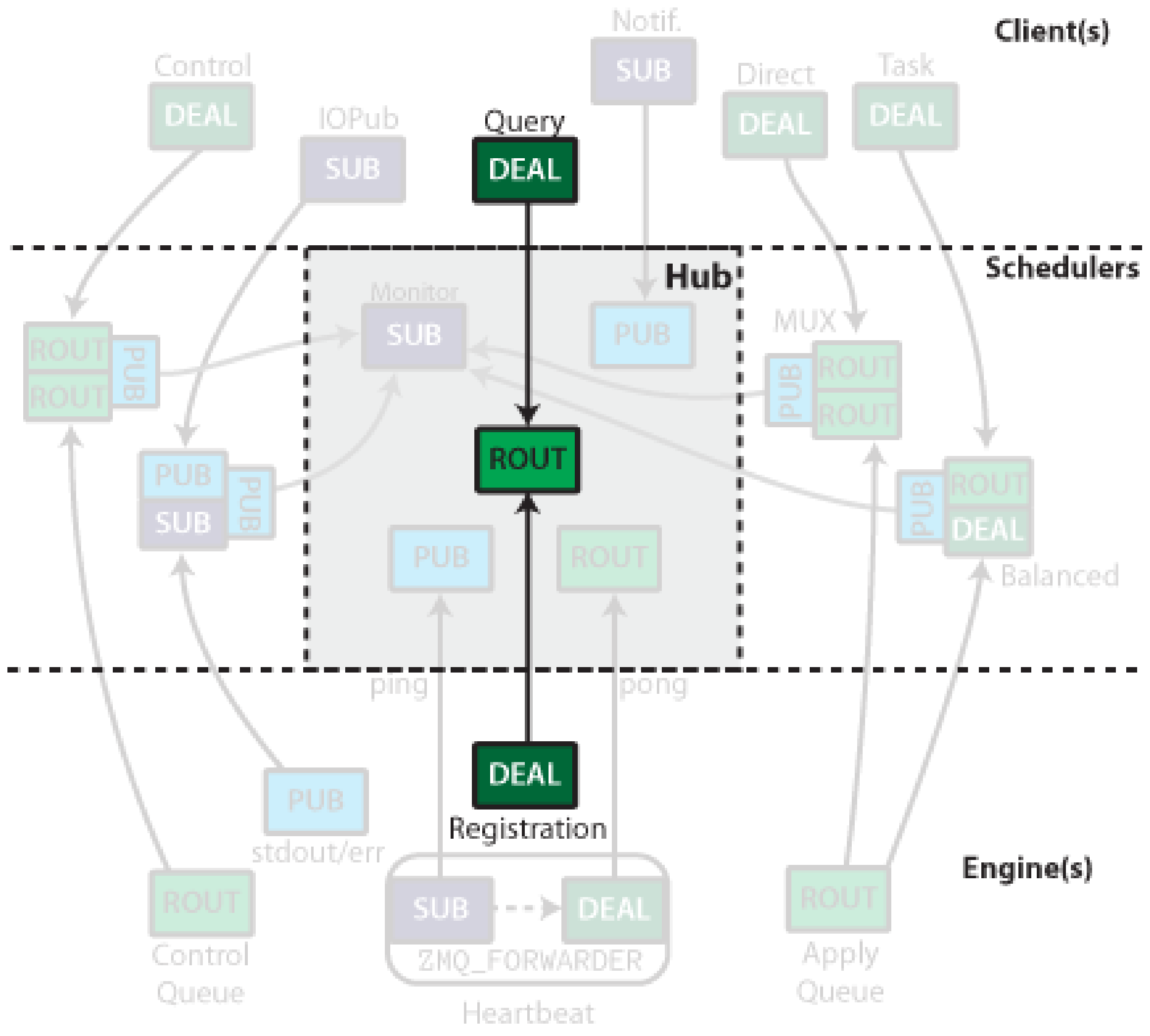


Fig. 7.2: Engines and Clients only need to know where the Query `ROUTER` is located to start connecting.

Registration

Once a controller is launched, the only information needed for connecting clients and/or engines is the IP/port of the Hub's `ROUTER` 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

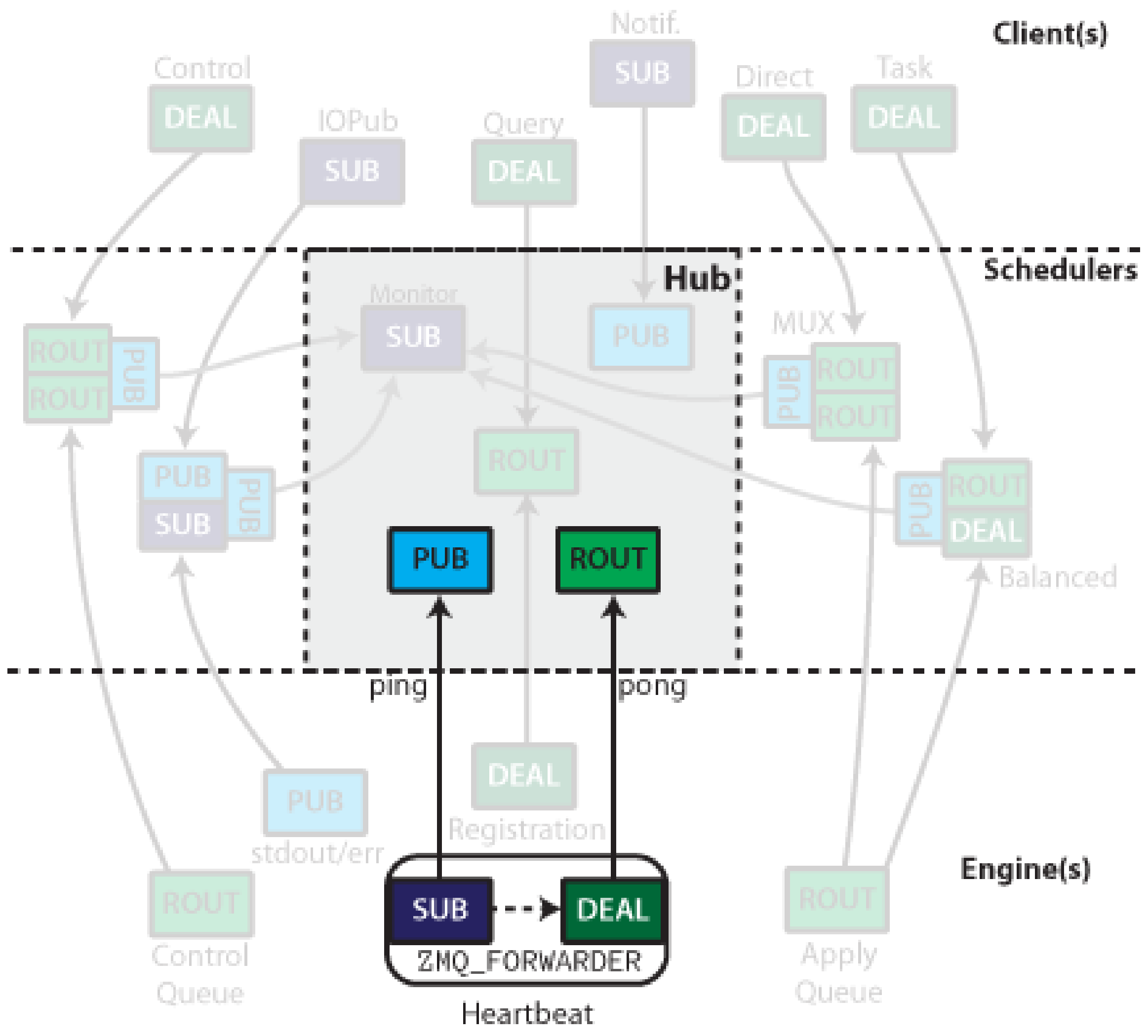


Fig. 7.3: The heartbeat sockets.

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 `DEALER` socket for output. The `SUB` socket is connected to the `PUB` socket labeled *ping*, and the DEALER is connected to the ROUTER labeled *pong*. This results in the same message being relayed back to the Heartbeat Monitor with the addition of the DEALER prefix. The Heartbeat Monitor receives all the replies via an ROUTER socket, and identifies which hearts are still beating by the zmq.IDENTITY prefix of the DEALER sockets, which information the Hub uses to notify clients of any changes in the available engines.

Schedulers

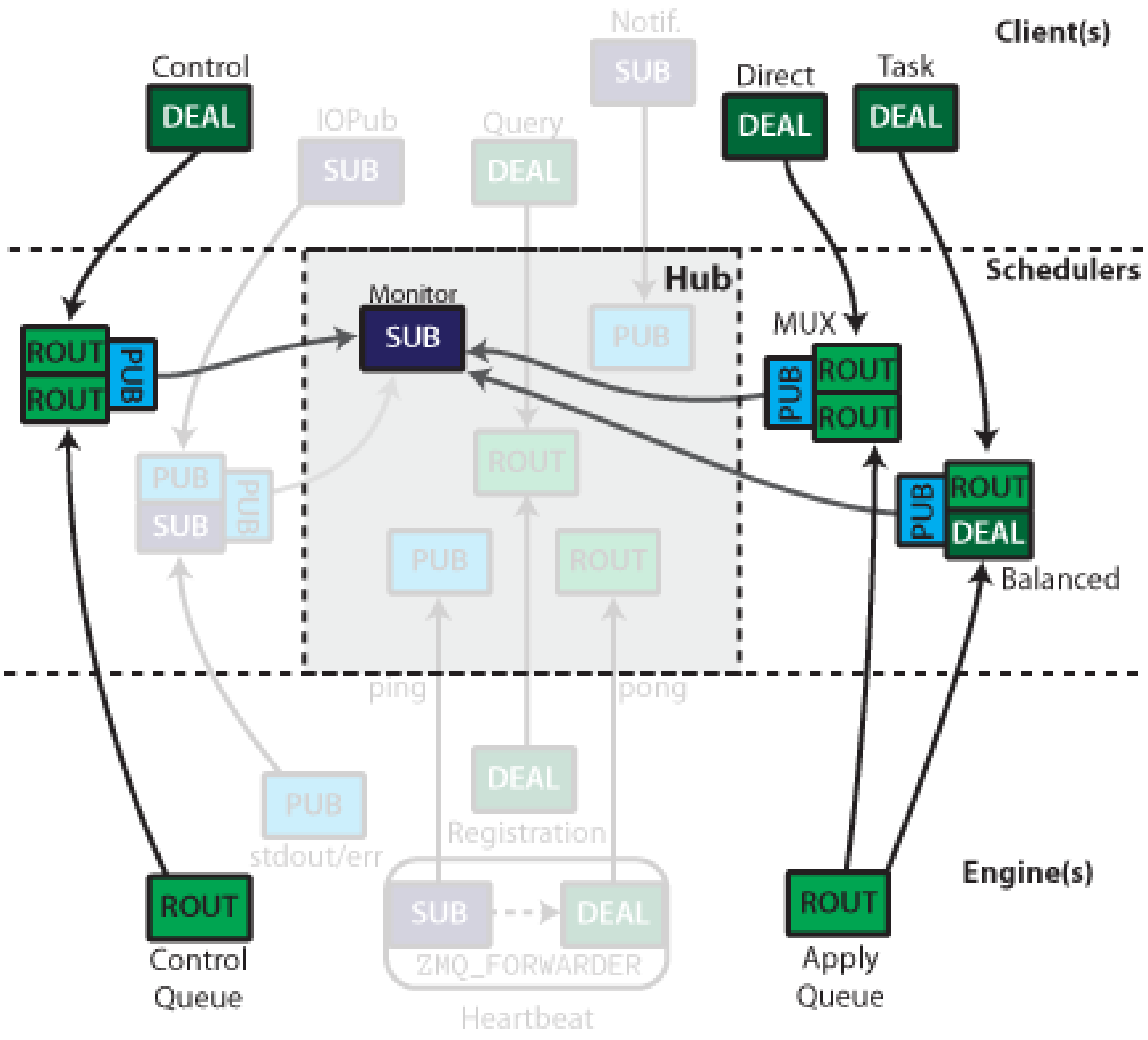


Fig. 7.4: Control message scheduler on the left, execution (apply) schedulers on the right.

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 `ROUTER` socket from the client side are assigned to an engine via ZMQ's `DEALER` 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, `ROUTER` is used for both in and output sockets in the device. Clients must specify the destination by the `zmq.IDENTITY` of the `ROUTER` socket connected to the downstream end of the device.

At the Kernel level, both of these `ROUTER` 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

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

Client connections

The hub's registrar `ROUTER` socket also listens for queries from clients as to queue status, and control instructions. Clients connect to this socket via an `DEALER` 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.

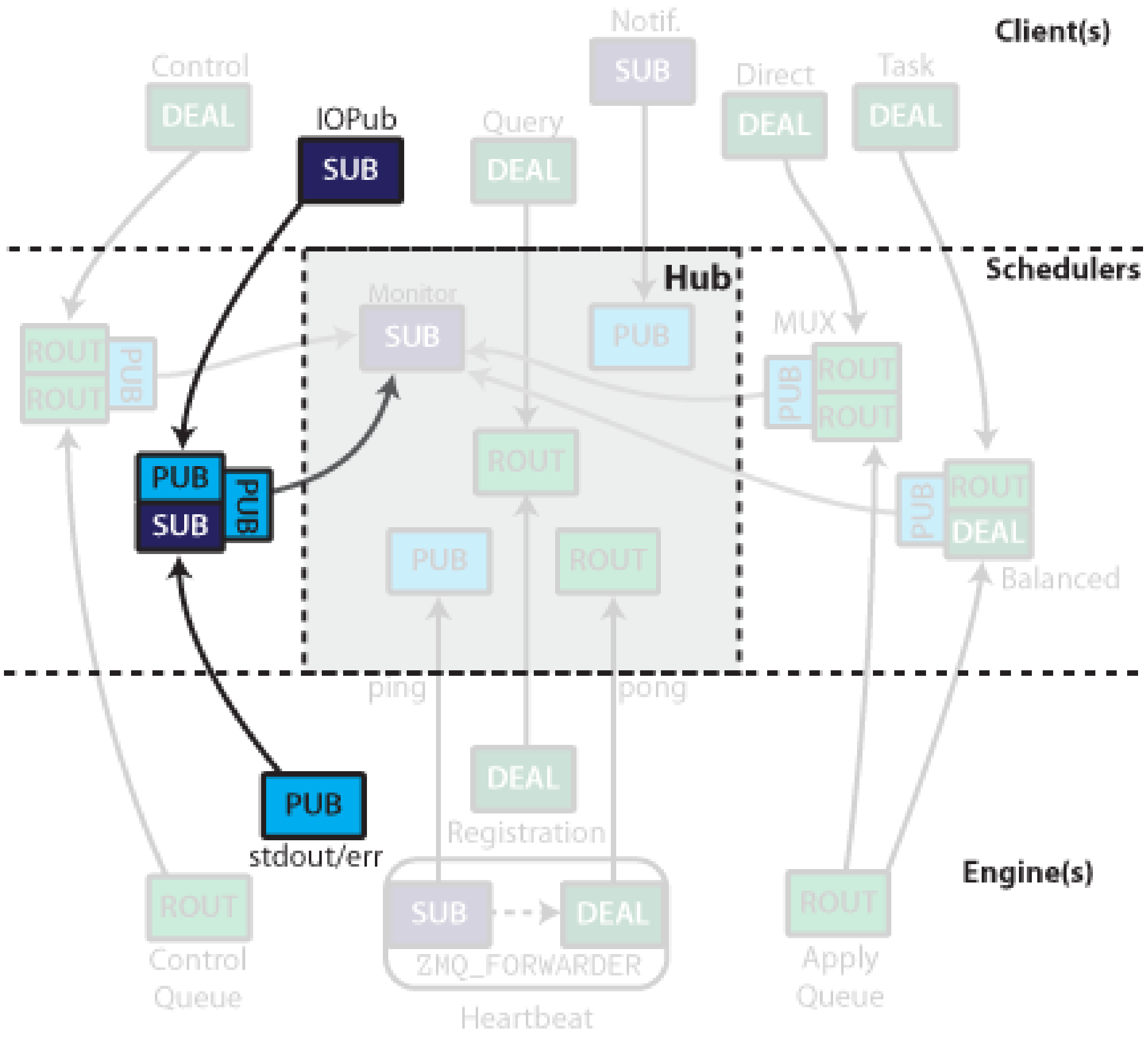


Fig. 7.5: stdout/err are published via a `PUB/SUB` MonitoredQueue

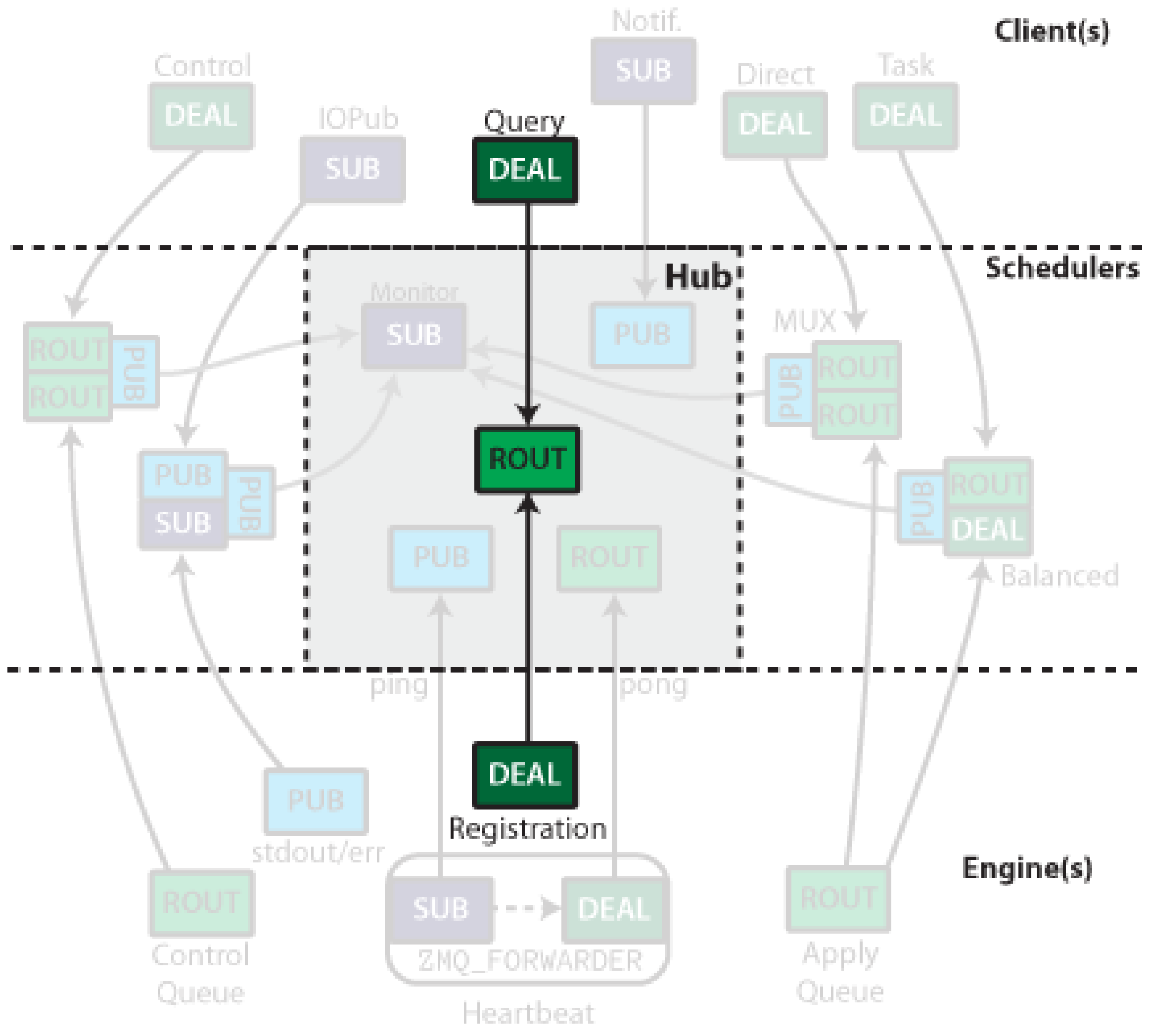


Fig. 7.6: Clients connect to an `ROUTER` socket to query the hub.

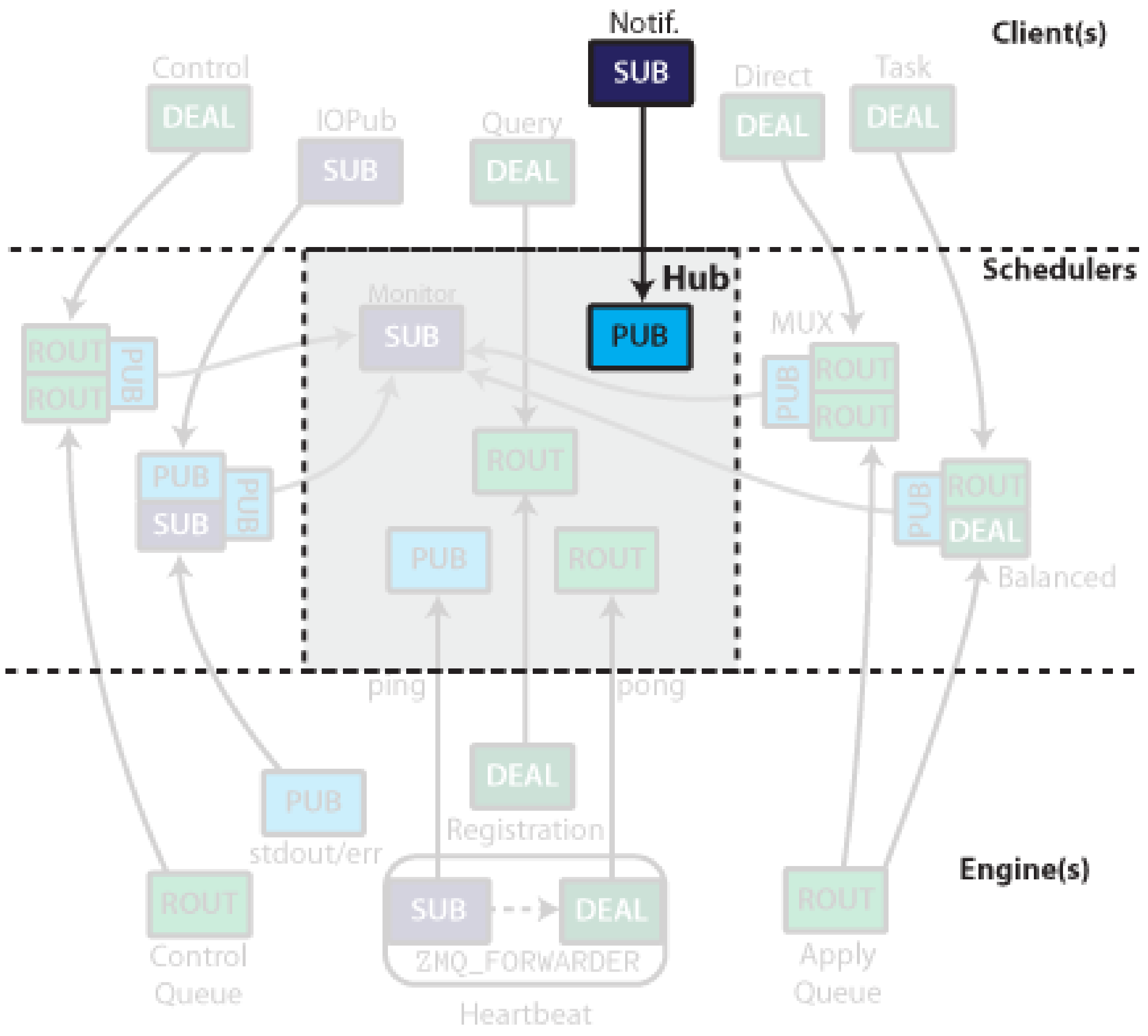


Fig. 7.7: Engine registration events are published via a `PUB` socket.

CHAPTER 8

The IPython API

CHAPTER 9

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, IPython notebook.
- **Benjamin (Min) Ragan-Kelley** <benjaminrk-AT-gmail.com> Parallel computing infrastructure, IPython core, IPython notebook.
- **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.
- **Paul Ivanov** <pi-AT-berkeley.edu> IPython core, documentation.

- **Matthias Bussonnier** <bussonniermatthias-AT-gmail.com> IPython notebook, nbviewer, nbconvert.
- **Julian Taylor** <jtaylor.debian-AT-googlemail.com> IPython core, Debian packaging.
- **Brad Froehle** <brad.froehle-AT-gmail.com> IPython core.

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 Riciputi <andrea.riciputi-AT-libero.it> Mac OSX information, Fink package management.
- Gary Bishop <gb-AT-cs.unc.edu> Bug reports, and patches to work around the exception handling idiosyncracies of WxPython. Readline and color support for Windows.
- Jeffrey Collins <Jeff.Collins-AT-vexcel.com>. Bug reports. Much improved readline support, including fixes for Python 2.3.
- Dryice Liu <dryice-AT-liu.com.cn> FreeBSD port.

- Mike Heeter <korora-AT-SDF.LONESTAR.ORG>
- Christopher Hart <hart-AT-caltech.edu> PDB integration.
- Milan Zamazal <pdm-AT-zamazal.org> Emacs info.
- Philip Hisley <compsys-AT-starpower.net>
- Holger Krekel <pyth-AT-devel.trillke.net> Tab completion, lots more.
- Robin Siebler <robinsiebler-AT-starband.net>
- Ralf Ahlbrink <ralf_ahlbrink-AT-web.de>
- Thorsten Kampe <thorsten-AT-thorstenkampe.de>
- Fredrik Kant <fredrik.kant-AT-front.com> Windows setup.
- Syver Enstad <syver-en-AT-online.no> Windows setup.
- Richard <rxe-AT-renre-europe.com> Global embedding.
- Hayden Callow <h.callow-AT-elec.canterbury.ac.nz> Gnuplot.py 1.6 compatibility.
- Leonardo Santagada <retype-AT-terra.com.br> Fixes for Windows installation.
- Christopher Armstrong <radix-AT-twistedmatrix.com> Bugfixes.
- Francois Pinard <pinard-AT-iro.umontreal.ca> Code and documentation fixes.
- Cory Dodt <cdodt-AT-fcoe.k12.ca.us> Bug reports and Windows ideas. Patches for Windows installer.
- Olivier Aubert <oaubert-AT-bat710.univ-lyon1.fr> New magics.
- King C. Shu <kingshu-AT-myrealbox.com> Autoindent patch.
- Chris Drexler <chris-AT-ac-drexler.de> Readline packages for Win32/CygWin.
- Gustavo Cordova Avila <gcordova-AT-sismex.com> EvalDict code for nice, lightweight string interpolation.
- Kasper Souren <Kasper.Souren-AT-ircam.fr> Bug reports, ideas.
- Gever Tulley <gever-AT-helium.com> Code contributions.
- Ralf Schmitt <ralf-AT-brainbot.com> Bug reports & fixes.
- Oliver Sander <osander-AT-gmx.de> Bug reports.
- Rod Holland <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 IPython, including making necessary changes to matplotlib itself.
- Matthew Arnison <maffew-AT-cat.org.au> Bug reports, '%run -d' idea.
- Prabhu Ramachandran <prabhu_r-AT-users.sourceforge.net> Help with (X)Emacs support, threading patches, ideas...
- Norbert Tretkowski <tretkowski-AT-inittab.de> help with Debian packaging and distribution.
- George Sakkis <gsakkis-AT-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-ml1.net> Official OpenBSD port.
- Ondrej Certik <ondrej-AT-certik.cz> Set up the IPython docs to use the new Sphinx system used by Python, Matplotlib and many more projects.
- Stefan van der Walt <stefan-AT-sun.ac.za> Design and prototype of the Traits based config system.

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:

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
- Paul Ivanov
- Evan Patterson
- Fernando Perez (project leader)
- Benjamin Ragan-Kelley
- Ville M. Vainio
- Gael Varoququx
- 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

[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>

Python Module Index

Index