IPython Documentation

Release 0.9.1

The IPython Development Team

September 14, 2008

CONTENTS

CHAPTER

ONE

Introduction

1.1 Overview

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

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

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

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

1.2 Enhanced interactive Python shell

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

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

1.2.1 Main features of the interactive shell

- Dynamic object introspection. One can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke ('?', and using '??' provides additional detail).
- Searching through modules and namespaces with '*' wildcards, both when using the '?' system and via the '%psearch' command.
- Completion in the local namespace, by typing TAB at the prompt. This works for keywords, modules, methods, variables and files in the current directory. This is supported via the readline library, and full access to configuring readline's behavior is provided. Custom completers can be implemented easily for different purposes (system commands, magic arguments etc.)
- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.
- User-extensible 'magic' commands. A set of commands prefixed with '%' is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.
- Alias facility for defining your own system aliases.
- Complete system shell access. Lines starting with '!' are passed directly to the system shell, and using '!!' or 'var = !cmd' captures shell output into python variables for further use.
- Background execution of Python commands in a separate thread. IPython has an internal job manager called jobs, and a convenience backgrounding magic function called '%bg'.
- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with '$' is expanded. A double '$$' allows passing a literal '$' to the shell (for access to shell and environment variables like **PATH**).
- Filesystem navigation, via a magic '%cd' command, along with a persistent bookmark system (using '%bookmark') for fast access to frequently visited directories.
- A lightweight persistence framework via the '%store' command, which allows you to save arbitrary Python variables. These get restored automatically when your session restarts.
- Automatic indentation (optional) of code as you type (through the readline library).
- Macro system for quickly re-executing multiple lines of previous input with a single name. Macros can be stored persistently via '%store' and edited via '%edit'.
- Session logging (you can then later use these logs as code in your programs). Logs can optionally timestamp all input, and also store session output (marked as comments, so the log remains valid Python source code).
- Session restoring: logs can be replayed to restore a previous session to the state where you left it.
- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information (basically a terminal version of the cgitb module).

- Auto-parentheses: callable objects can be executed without parentheses: '`sin 3`' is automatically converted to '`sin(3)`'.
- Auto-quoting: using '`,`', or '`;`' as the first character forces auto-quoting of the rest of the line: '`,my_function a b`' becomes automatically '`my_function("a","b")`', while '`;my_function a b`' becomes '`my_function("a b")`'.
- Extensible input syntax. You can define filters that pre-process user input to simplify input in special situations. This allows for example pasting multi-line code fragments which start with '`>>>`' or '`...`' such as those from other python sessions or the standard Python documentation.
- Flexible configuration system. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.
- Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).
- Easy debugger access. You can set IPython to call up an enhanced version of the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The '`%run`' magic command (with the '`-d`' option) can run any script under pdb's control, automatically setting initial breakpoints for you. This version of pdb has IPython-specific improvements, including tab-completion and traceback coloring support. For even easier debugger access, try '`%debug`' after seeing an exception. winpdb is also supported, see ipy_winpdb extension.
- Profiler support. You can run single statements (similar to '`profile.run()`') or complete programs under the profiler's control. While this is possible with standard cProfile or profile modules, IPython wraps this functionality with magic commands (see '`%prun`' and '`%run -p`') convenient for rapid interactive work.
- Doctest support. The special '`%doctest_mode`' command toggles a mode that allows you to paste existing doctests (with leading '`>>>`' prompts and whitespace) and uses doctest-compatible prompts and output, so you can use IPython sessions as doctest code.

1.3 Interactive parallel computing

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

The main features of this system are:

- Quickly parallelize Python code from an interactive Python/IPython session.
- A flexible and dynamic process model that be deployed on anything from multicore workstations to supercomputers.

- An architecture that supports many different styles of parallelism, from message passing to task farming. And all of these styles can be handled interactively.
- Both blocking and fully asynchronous interfaces.
- High level APIs that enable many things to be parallelized in a few lines of code.
- Write parallel code that will run unchanged on everything from multicore workstations to supercomputers.
- Full integration with Message Passing libraries (MPI).
- Capabilities based security model with full encryption of network connections.
- Share live parallel jobs with other users securely. We call this collaborative parallel computing.
- Dynamically load balanced task farming system.
- Robust error handling. Python exceptions raised in parallel execution are gathered and presented to the top-level code.

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

1.3.1 Portability and Python requirements

As of the 0.9 release, IPython requires Python 2.4 or greater. We have not begun to test IPython on Python 2.6 or 3.0, but we expect it will work with some minor changes.

IPython is known to work on the following operating systems:

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

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

CHAPTER

TWO

Installation

2.1 Overview

This document describes the steps required to install IPython. 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 requires Python version 2.4 or greater. We have not tested IPython with the upcoming 2.6 or 3.0 versions.

Warning: IPython will not work with Python 2.3 or below.

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

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

2.2 Installing IPython itself

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

2.2.1 Installation using easy_install

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

```
$ easy_install IPython
```

That's it.

2.2.2 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**.

2.2.3 Windows

There are a few caveats for Windows users. The main issue is that a basic `python setup.py install` approach won't create `.bat` file or Start Menu shortcuts, which most users want. To get an installation with these, there are two choices:

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

2.2.4 Installing the development version

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

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

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

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

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

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

```
$ bzr pull
```

2.3 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.

2.3.1 readline

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

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

On OS X, the built-in Python doesn't not have `readline` because of license issues. Starting with OS X 10.5 (Leopard), Apple's built-in Python has a BSD-licensed not-quite-compatible readline replacement. As of IPython 0.9, many of the issues related to the differences between readline and libedit have been resolved. For many users, libedit may be sufficient.

Most users on OS X will want to get the full `readline` module. To get a working `readline` module, just do (with `setuptools` installed):

```
$ easy_install readline
```

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

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

2.3.2 nose

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

```
$ easy_install nose
```

Another way of getting this is to do:

```
$ easy_install IPython[test]
```

For more installation options, see the nose website. Once you have nose installed, you can run IPython's test suite using the iptest command:

```
$ iptest
```

2.3.3 pexpect

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

```
$ easy_install pexpect
```

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

2.4 Dependencies for IPython.kernel (parallel computing)

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

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

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

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

2.4.1 zope.interface and Twisted

On Unix style platforms (including OS X), the simplest way of getting the these is to use **easy_install**:

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

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

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

2.4.2 Foolscap

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

On all platforms a simple:

```
$ easy_install foolscap
```

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

2.4.3 pyOpenSSL

IPython requires an older version of pyOpenSSL (0.6 rather than the current 0.7). There are a couple of options for getting this:

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

2.5 Dependencies for IPython.frontend (the IPython GUI)

2.5.1 wxPython

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

CHAPTER

THREE

Using IPython for interactive work

3.1 Quick IPython tutorial

Contents

IPython can be used as an improved replacement for the Python prompt, and for that you don't really need to read any more of this manual. But in this section we'll try to summarize a few tips on how to make the

most effective use of it for everyday Python development, highlighting things you might miss in the rest of the manual (which is getting long). We'll give references to parts in the manual which provide more detail when appropriate.

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

3.1.1 Highlights

Tab completion

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

Explore your objects

Typing object_name? will print all sorts of details about any object, including docstrings, function definition lines (for call arguments) and constructor details for classes. The magic commands %pdoc, %pdef, %psource and %pfile will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found). If automagic is on (it is by default), you don't need to type the '%' explicitly. See *this section* for more.

The *%run* magic command

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

Debug a Python script

Use the Python debugger, pdb. The %pdb command allows you to toggle on and off the automatic invocation of an IPython-enhanced pdb debugger (with coloring, tab completion and more) at any uncaught exception. The advantage of this is that pdb starts inside the function where the exception occurred, with all data still available. You can print variables, see code, execute statements and even walk up and down the call stack to track down the true source of the problem (which often is many layers in the stack above where the exception gets triggered). Running programs with %run and pdb active can be an efficient to develop and debug code, in many cases eliminating the need for print statements or external debugging tools. I often simply put a 1/0 in a place where I want to take a look so that pdb gets called, quickly view whatever variables I need to or

test various pieces of code and then remove the 1/0. Note also that '%run -d' activates pdb and automatically sets initial breakpoints for you to step through your code, watch variables, etc. The *output caching section* has more details.

Use the output cache

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

Suppress output

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

Input cache

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

Use your input history

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

Define your own system aliases

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

Call system shell commands

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

Use Python variables when calling the shell

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

Use profiles

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

Embed IPython in your programs

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

Use the Python profiler

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

Use IPython to present interactive demos

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

Run doctests

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

3.1.2 Source code handling tips

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

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

If you have typed various commands during a session, which you'd like to reuse, IPython provides you with a number of tools. Start by using %hist to see your input history, so you can see the line numbers of all input. Let us say that you'd like to reuse lines 10 through 20, plus lines 24 and 28. All the commands below can operate on these with the syntax:

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

where the command given can be:

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

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

3.1.3 Lightweight 'version control'

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

```
In [1]: %edit

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

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

hello - this is a temporary file

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

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

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

```
In [2]: edit -p

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

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

hello - now I made some changes

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

In [3]: edit _1

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

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

hello - this is a temporary file

IPython version control at work :)

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

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

3.1.4 Effective logging

A very useful suggestion sent in by Robert Kern follows:

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

```
include ipythonrc

# cancel earlier logfile invocation:

logfile ''

execute import time

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

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

I also added a shell alias for convenience:

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

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

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

3.2 IPython reference

Contents

3.2.1 Command-line usage

You start IPython with the command:

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

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

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

Special Threading Options

The following special options are ONLY valid at the beginning of the command line, and not later. This is because they control the initial- ization of ipython itself, before the normal option-handling mechanism is active.

-gthread, -qthread, -q4thread, -wthread, -pylab: Only one of these can be given, and it can only be given as the first option passed to IPython (it will have no effect in any other position). They provide threading support for the GTK, Qt (versions 3 and 4) and WXPython toolkits, and for the matplotlib library.

With any of the first four options, IPython starts running a separate thread for the graphical toolkit's operation, so that you can open and control graphical elements from within an IPython command line, without blocking. All four provide essentially the same functionality, respectively for GTK, Qt3, Qt4 and WXWidgets (via their Python interfaces).

Note that with -wthread, you can additionally use the -wxversion option to request a specific version of wx to be used. This requires that you have the wxversion Python module installed, which is part of recent wxPython distributions.

If -pylab is given, IPython loads special support for the mat plotlib library (http://matplotlib.sourceforge.net), allowing interactive usage of any of its backends as defined in the user's ~/.matplotlib/matplotlibrc file. It automatically activates GTK, Qt or WX threading for IPyhton if the choice of matplotlib backend requires it. It also modifies the %run command to correctly execute (without blocking) any matplotlib-based script which calls show() at the end.

-tk The -g/q/q4/wthread options, and -pylab (if matplotlib is configured to use GTK, Qt3, Qt4 or WX), will normally block Tk graphical interfaces. This means that when either GTK, Qt or WX threading is active, any attempt to open a Tk GUI will result in a dead window, and possibly cause the Python

interpreter to crash. An extra option, -tk, is available to address this issue. It can only be given as a second option after any of the above (-gthread, -wthread or -pylab).

If -tk is given, IPython will try to coordinate Tk threading with GTK, Qt or WX. This is however potentially unreliable, and you will have to test on your platform and Python configuration to determine whether it works for you. Debian users have reported success, apparently due to the fact that Debian builds all of Tcl, Tk, Tkinter and Python with pthreads support. Under other Linux environments (such as Fedora Core 2/3), this option has caused random crashes and lockups of the Python interpreter. Under other operating systems (Mac OSX and Windows), you'll need to try it to find out, since currently no user reports are available.

There is unfortunately no way for IPython to determine at run time whether -tk will work reliably or not, so you will need to do some experiments before relying on it for regular work.

Regular Options

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

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

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

-help print a help message and exit.

-pylab this can only be given as the first option passed to IPython (it will have no effect in any other position). It adds special support for the matplotlib library (http://matplotlib.sourceforge.ne), allowing interactive usage of any of its backends as defined in the user's .matplotlibrc file. It automatically activates GTK or WX threading for IPyhton if the choice of matplotlib backend requires it. It also modifies the %run command to correctly execute (without blocking) any matplotlib-based script which calls show() at the end. See Matplotlib support for more details.

-autocall <val> Make IPython automatically call any callable object even if you didn't type explicit parentheses. For example, 'str 43' becomes 'str(43)' automatically. The value can be '0' to disable the feature, '1' for smart autocall, where it is not applied if there are no more arguments on the line, and '2' for full autocall, where all callable objects are automatically called (even if no arguments are present). The default is '1'.

-[no]autoindent Turn automatic indentation on/off.

-[no]automagic make magic commands automatic (without needing their first character to be %). Type %magic at the IPython prompt for more information.

-[no]autoedit_syntax When a syntax error occurs after editing a file, automatically open the file to the trouble causing line for convenient fixing.

-[no]banner Print the initial information banner (default on).

-c <command> execute the given command string. This is similar to the -c option in the normal Python interpreter.

-cache_size, cs <n> size of the output cache (maximum number of entries to hold in memory). The default is 1000, you can change it permanently in your config file. Setting it to 0 completely disables the caching system, and the minimum value accepted is 20 (if you provide a value less than 20, it is reset to 0 and a warning is issued) This limit is defined because otherwise you'll spend more time re-flushing a too small cache than working.

-classic, cl Gives IPython a similar feel to the classic Python prompt.

-colors <scheme> Color scheme for prompts and exception reporting. Currently implemented: NoColor, Linux and LightBG.

-[no]color_info IPython can display information about objects via a set of functions, and optionally can use colors for this, syntax highlighting source code and various other elements. However, because this information is passed through a pager (like 'less') and many pagers get confused with color codes, this option is off by default. You can test it and turn it on permanently in your ipythonrc file if it works for you. As a reference, the 'less' pager supplied with Mandrake 8.2 works ok, but that in RedHat 7.2 doesn't.

Test it and turn it on permanently if it works with your system. The magic function %color_info allows you to toggle this interactively for testing.

-[no]debug Show information about the loading process. Very useful to pin down problems with your configuration files or to get details about session restores.

-[no]deep_reload: IPython can use the deep_reload module which reloads changes in modules recursively (it replaces the reload() function, so you don't need to change anything to use it). deep_reload() forces a full reload of modules whose code may have changed, which the default reload() function does not.

When deep_reload is off, IPython will use the normal reload(), but deep_reload will still be available as dreload(). This feature is off by default [which means that you have both normal reload() and dreload()].

-editor <name> Which editor to use with the %edit command. By default, IPython will honor your EDITOR environment variable (if not set, vi is the Unix default and notepad the Windows one). Since this editor is invoked on the fly by IPython and is meant for editing small code snippets, you may want to use a small, lightweight editor here (in case your default EDITOR is something like Emacs).

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

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

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

-logplay, lp <name>

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

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

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

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

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

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

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

-profile, p <name>

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

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

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

-prompt_in1, pi1 <string>

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

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

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

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

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

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

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

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

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

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

-separate_in, si <string>

separator before input prompts. Default: 'n'

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

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

-nosep shorthand for '-SeparateIn 0 -SeparateOut 0 -SeparateOut2 0'. Simply removes all input/output separators.

-upgrade allows you to upgrade your IPYTHONDIR configuration when you install a new version of IPython. Since new versions may include new command line options or example files, this copies updated ipythonrc-type files. However, it backs up (with a .old extension) all files which it overwrites so that you can merge back any customizations you might have in your personal files. Note that you should probably use %upgrade instead, it's a safer alternative.

-Version print version information and exit.

-wxversion <string> Select a specific version of wxPython (used in conjunction with -wthread). Requires the wxversion module, part of recent wxPython distributions

-xmode <modename>

Mode for exception reporting.

Valid modes: Plain, Context and Verbose.

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

3.2.2 Interactive use

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

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

Caution for Windows users

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

Magic command system

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

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

If you have 'automagic' enabled (in your ipythonrc file, via the command line option -automagic or with the %automagic function), you don't need to type in the % explicitly. IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type 'cd mydir' to go to directory 'mydir'. The automagic system has the lowest possible precedence in name searches, so defining

an identifier with the same name as an existing magic function will shadow it for automagic use. You can still access the shadowed magic function by explicitly using the % character at the beginning of the line.

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

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

/home/fperez/ipython

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

In [3]: cd .. # and doesn't work as a function anymore

------------------------------

    File "<console>", line 1

      cd ..

         ^

SyntaxError: invalid syntax

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

/home/fperez

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

In [6]: cd ipython # automagic can work again

/home/fperez/ipython
```

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

```
import IPython.ipapi

ip = IPython.ipapi.get()

def doimp(self, arg):

    ip = self.api

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

    arg,arg,arg)

    )

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

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

execute __IP.magic_cl = __IP.magic_clear

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

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

Magic commands

The rest of this section is automatically generated for each release from the docstrings in the IPython code. Therefore the formatting is somewhat minimal, but this method has the advantage of having information always in sync with the code.

A list of all the magic commands available in IPython's default installation follows. This is similar to what you'll see by simply typing %magic at the prompt, but that will also give you information about magic commands you may have added as part of your personal customizations.

%Exit:

```
Exit IPython without confirmation.
```

%Pprint:

```
Toggle pretty printing on/off.
```

%alias:

```
Define an alias for a system command.

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

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

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

You can use the %l specifier in an alias definition to represent the
whole line when the alias is called.  For example:

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

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

  In [1]: alias parts echo first %s second %s\
  In [2]: %parts A B\
  first A second B\
```

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

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

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

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

You can use the alias facility to acess all of $PATH.  See the %rehash
and %rehashx functions, which automatically create aliases for the
contents of your $PATH.

If called with no parameters, %alias prints the current alias table.
```

%autocall:

```
Make functions callable without having to type parentheses.

Usage:

   %autocall [mode]

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

In more detail, these values mean:

0 -> fully disabled

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

In this mode, you get:

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

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

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

In [4]: callable
------> callable()

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

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

%autoindent:

```
Toggle autoindent on/off (if available).
```

%automagic:

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

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

 - on,1,True: to activate

 - off,0,False: to deactivate.

Note that magic functions have lowest priority, so if there's a
variable whose name collides with that of a magic fn, automagic won't
work for that function (you get the variable instead). However, if you
delete the variable (del var), the previously shadowed magic function
becomes visible to automagic again.
```

%bg:

```
Run a job in the background, in a separate thread.

For example,

  %bg myfunc(x,y,z=1)

will execute 'myfunc(x,y,z=1)' in a background thread.  As soon as the
execution starts, a message will be printed indicating the job
number.  If your job number is 5, you can use

  myvar = jobs.result(5)  or  myvar = jobs[5].result

to assign this result to variable 'myvar'.
```

```
IPython has a job manager, accessible via the 'jobs' object.  You can
type jobs? to get more information about it, and use jobs.<TAB> to see
its attributes.  All attributes not starting with an underscore are
meant for public use.

In particular, look at the jobs.new() method, which is used to create
new jobs.  This magic %bg function is just a convenience wrapper
around jobs.new(), for expression-based jobs.  If you want to create a
new job with an explicit function object and arguments, you must call
jobs.new() directly.

The jobs.new docstring also describes in detail several important
caveats associated with a thread-based model for background job
execution.  Type jobs.new? for details.

You can check the status of all jobs with jobs.status().

The jobs variable is set by IPython into the Python builtin namespace.
If you ever declare a variable named 'jobs', you will shadow this
name.  You can either delete your global jobs variable to regain
access to the job manager, or make a new name and assign it manually
to the manager (stored in IPython's namespace).  For example, to
assign the job manager to the Jobs name, use:

  Jobs = __builtins__.jobs
```

%bookmark:

```
Manage IPython's bookmark system.

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

You can later on access a bookmarked folder with:
  %cd -b <name>
or simply '%cd <name>' if there is no directory called <name> AND
there is such a bookmark defined.

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

%cd:

```
Change the current working directory.

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

```
Usage:

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

  cd -: changes to the last visited directory.

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

  cd -b <bookmark_name>: jump to a bookmark set by %bookmark
     (note: cd <bookmark_name> is enough if there is no
      directory <bookmark_name>, but a bookmark with the name exists.)
      'cd -b <tab>' allows you to tab-complete bookmark names.

Options:

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

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

%clear:

```
     Clear various data (e.g. stored history data)

%clear out - clear output history
%clear in  - clear input history
%clear shadow_compress - Compresses shadow history (to speed up ipython)
%clear shadow_nuke - permanently erase all entries in shadow history
%clear dhist - clear dir history
```

%color_info:

```
Toggle color_info.

The color_info configuration parameter controls whether colors are
used for displaying object details (by things like %psource, %pfile or
the '?' system). This function toggles this value with each call.

Note that unless you have a fairly recent pager (less works better
than more) in your system, using colored object information displays
will not work properly. Test it and see.
```

%colors:

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

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.
```

%cpaste:

```
Allows you to paste & execute a pre-formatted code block from clipboard

You must terminate the block with '--' (two minus-signs) alone on the
line. You can also provide your own sentinel with '%paste -s %%' ('%%'
is the new sentinel for this operation)

The block is dedented prior to execution to enable execution of method
definitions. '>' and '+' characters at the beginning of a line are
ignored, to allow pasting directly from e-mails or diff files. The
executed block is also assigned to variable named 'pasted_block' for
later editing with '%edit pasted_block'.

You can also pass a variable name as an argument, e.g. '%cpaste foo'.
This assigns the pasted block to variable 'foo' as string, without
dedenting or executing it.

Do not be alarmed by garbled output on Windows (it's a readline bug).
Just press enter and type -- (and press enter again) and the block
will be what was just pasted.

IPython statements (magics, shell escapes) are not supported (yet).
```

%debug:

```
Activate the interactive debugger in post-mortem mode.

If an exception has just occurred, this lets you inspect its stack
frames interactively.  Note that this will always work only on the last
traceback that occurred, so you must call this quickly after an
exception that you wish to inspect has fired, because if another one
occurs, it clobbers the previous one.

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

%dhist:

```
Print your history of visited directories.

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

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

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

%dirs:

```
Return the current directory stack.
```

%doctest_mode:

```
Toggle doctest mode on and off.

This mode allows you to toggle the prompt behavior between normal
IPython prompts and ones that are as similar to the default IPython
interpreter as possible.

It also supports the pasting of code snippets that have leading '>>>'
and '...' prompts in them.  This means that you can paste doctests from
files or docstrings (even if they have leading whitespace), and the
code will execute correctly.  You can then use '%history -tn' to see
the translated history without line numbers; this will give you the
input after removal of all the leading prompts and whitespace, which
can be pasted back into an editor.

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

%ed:

```
Alias to %edit.
```

%edit:

```
Bring up an editor and execute the resulting code.

Usage:
  %edit [options] [args]

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

You can also set the value of this editor via the command line option
'-editor' or in your ipythonrc file. This is useful if you wish to use
specifically for IPython an editor different from your typical default
(and for Windows users who typically don't set environment variables).

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

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

```
Options:

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

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

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

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


Arguments:

If arguments are given, the following possibilites exist:

- The arguments are numbers or pairs of colon-separated numbers (like
1 4:8 9). These are interpreted as lines of previous input to be
loaded into the editor. The syntax is the same of the %macro command.

- If the argument doesn't start with a number, it is evaluated as a
variable and its contents loaded into the editor. You can thus edit
any string which contains python code (including the result of
previous edits).

- If the argument is the name of an object (other than a string),
IPython will try to locate the file where it was defined and open the
editor at the point where it is defined. You can use `%edit function`
to load an editor exactly at the point where 'function' is defined,
edit it and have the file be executed automatically.

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

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

```
- If the argument is not found as a variable, IPython will look for a
file with that name (adding .py if necessary) and load it into the
editor. It will execute its contents with execfile() when you exit,
loading any code in the file into your interactive namespace.

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

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

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

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

We can then call the function foo():

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

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

In [3]: ed foo\
Editing... done. Executing edited code...

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

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

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

In [8]: ed\
Editing... done. Executing edited code...\
hello\
Out[8]: "print 'hello'\n"

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

In [9]: ed _\
Editing... done. Executing edited code...\
hello world\
Out[9]: "print 'hello world'\n"

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

```
In [10]: ed _8\
Editing... done. Executing edited code...\
hello again\
Out[10]: "print 'hello again'\n"


Changing the default editor hook:

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

%env:

```
List environment variables.
```

%exit:

```
Exit IPython, confirming if configured to do so.

You can configure whether IPython asks for confirmation upon exit by
setting the confirm_exit flag in the ipythonrc file.
```

%hist:

```
Alternate name for %history.
```

%history:

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

%history       -> print at most 40 inputs (some may be multi-line)\
%history n     -> print at most n inputs\
%history n1 n2 -> print inputs between n1 and n2 (n2 not included)\

Each input's number <n> is shown, and is accessible as the
automatically generated variable _i<n>.  Multi-line statements are
printed starting at a new line for easy copy/paste.


Options:

  -n: do NOT print line numbers. This is useful if you want to get a
  printout of many lines which can be directly pasted into a text
  editor.

  This feature is only available if numbered prompts are in use.
```

```
  -t: (default) print the 'translated' history, as IPython understands it.
  IPython filters your input and converts it all into valid Python source
  before executing it (things like magics or aliases are turned into
  function calls, for example). With this option, you'll see the native
  history instead of the user-entered version: '%cd /' will be seen as
  '_ip.magic("%cd /")' instead of '%cd /'.

  -r: print the 'raw' history, i.e. the actual commands you typed.

  -g: treat the arg as a pattern to grep for in (full) history.
  This includes the "shadow history" (almost all commands ever written).
  Use '%hist -g' to show full shadow history (may be very long).
  In shadow history, every index nuwber starts with 0.

  -f FILENAME: instead of printing the output to the screen, redirect it to
   the given file.  The file is always overwritten, though IPython asks for
   confirmation first if it already exists.
```

%logoff:

```
Temporarily stop logging.

You must have previously started logging.
```

%logon:

```
Restart logging.

This function is for restarting logging which you've temporarily
stopped with %logoff. For starting logging for the first time, you
must use the %logstart function, which allows you to specify an
optional log filename.
```

%logstart:

```
Start logging anywhere in a session.

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

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

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

%logstart takes a second optional parameter: logging mode. This can be one
of (note that the modes are given unquoted):\
  append: well, that says it.\
  backup: rename (if exists) to name~ and start name.\
  global: single logfile in your home dir, appended to.\
  over  : overwrite existing log.\
  rotate: create rotating logs name.1~, name.2~, etc.
```

```
Options:

  -o: log also IPython's output.  In this mode, all commands which
  generate an Out[NN] prompt are recorded to the logfile, right after
  their corresponding input line.  The output lines are always
  prepended with a '#[Out]# ' marker, so that the log remains valid
  Python code.

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

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

  -r: log 'raw' input.  Normally, IPython's logs contain the processed
  input, so that user lines are logged in their final form, converted
  into valid Python.  For example, %Exit is logged as
  '_ip.magic("Exit").  If the -r flag is given, all input is logged
  exactly as typed, with no transformations applied.

  -t: put timestamps before each input line logged (these are put in
  comments).
```

%logstate:

```
Print the status of the logging system.
```

%logstop:

```
Fully stop logging and close log file.

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

%lsmagic:

```
List currently available magic functions.
```

%macro:

```
Define a set of input lines as a macro for future re-execution.

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

Options:

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

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

The notation for indicating number ranges is: n1-n2 means 'use line
numbers n1,...n2' (the endpoint is included).  That is, '5-7' means
using the lines numbered 5,6 and 7.

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

For example, if your history contains (%hist prints it):

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

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

  In [51]: %macro my_macro 44-47 49

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

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

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

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

  'print macro_name'.

For one-off cases which DON'T contain magic function calls in them you
can obtain similar results by explicitly executing slices from your
input history with:

  In [60]: exec In[44:48]+In[49]
```

%magic:

```
Print information about the magic function system.
```

%mglob:

```
        This program allows specifying filenames with "mglob" mechanism.
Supported syntax in globs (wilcard matching patterns)::

 *.cpp ?ellowo*
     - obvious. Differs from normal glob in that dirs are not included.
       Unix users might want to write this as: "*.cpp" "?ellowo*"
 rec:/usr/share=*.txt,*.doc
     - get all *.txt and *.doc under /usr/share,
       recursively
 rec:/usr/share
     - All files under /usr/share, recursively
 rec:*.py
     - All .py files under current working dir, recursively
 foo
     - File or dir foo
 !*.bak readme*
     - readme*, exclude files ending with .bak
 !.svn/ !.hg/ !*_Data/ rec:.
     - Skip .svn, .hg, foo_Data dirs (and their subdirs) in recurse.
       Trailing / is the key, \ does not work!
 dir:foo
     - the directory foo if it exists (not files in foo)
 dir:*
     - all directories in current folder
 foo.py bar.* !h* rec:*.py
     - Obvious. !h* exclusion only applies for rec:*.py.
       foo.py is *not* included twice.
 @filelist.txt
     - All files listed in 'filelist.txt' file, on separate lines.
```

%page:

```
Pretty print the object and display it through a pager.

%page [options] OBJECT

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

Options:

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

%pdb:

```
Control the automatic calling of the pdb interactive debugger.

Call as '%pdb on', '%pdb 1', '%pdb off' or '%pdb 0'. If called without
```

```
argument it works as a toggle.

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

The initial state of this feature is set in your ipythonrc
configuration file (the variable is called 'pdb').

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

%pdef:

```
Print the definition header for any callable object.

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

%pdoc:

```
Print the docstring for an object.

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

%pfile:

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

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

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

%pinfo:

```
Provide detailed information about an object.

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

%popd:

```
Change to directory popped off the top of the stack.
```

%profile:

```
Print your currently active IPyhton profile.
```

%prun:

```
 Run a statement through the python code profiler.

 Usage:\
   %prun [options] statement

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

 Options:

 -l <limit>: you can place restrictions on what or how much of the
 profile gets printed. The limit value can be:

   * A string: only information for function names containing this string
   is printed.

   * An integer: only these many lines are printed.

   * A float (between 0 and 1): this fraction of the report is printed
   (for example, use a limit of 0.4 to see the topmost 40% only).

 You can combine several limits with repeated use of the option. For
 example, '-l __init__ -l 5' will print only the topmost 5 lines of
 information about class constructors.

 -r: return the pstats.Stats object generated by the profiling. This
 object has all the information about the profile in it, and you can
 later use it for further analysis or in other functions.

-s <key>: sort profile by given key. You can provide more than one key
 by using the option several times: '-s key1 -s key2 -s key3...'. The
 default sorting key is 'time'.

 The following is copied verbatim from the profile documentation
 referenced below:

 When more than one key is provided, additional keys are used as
 secondary criteria when the there is equality in all keys selected
 before them.

 Abbreviations can be used for any key names, as long as the
 abbreviation is unambiguous.  The following are the keys currently
 defined:

         Valid Arg       Meaning\
```

```
            "calls"      call count\
            "cumulative" cumulative time\
            "file"       file name\
            "module"     file name\
            "pcalls"     primitive call count\
            "line"       line number\
            "name"       function name\
            "nfl"        name/file/line\
            "stdname"    standard name\
            "time"       internal time

 Note that all sorts on statistics are in descending order (placing
 most time consuming items first), where as name, file, and line number
 searches are in ascending order (i.e., alphabetical). The subtle
 distinction between "nfl" and "stdname" is that the standard name is a
 sort of the name as printed, which means that the embedded line
 numbers get compared in an odd way.  For example, lines 3, 20, and 40
 would (if the file names were the same) appear in the string order
 "20" "3" and "40".  In contrast, "nfl" does a numeric compare of the
 line numbers.  In fact, sort_stats("nfl") is the same as
 sort_stats("name", "file", "line").

 -T <filename>: save profile results as shown on screen to a text
 file. The profile is still shown on screen.

 -D <filename>: save (via dump_stats) profile statistics to given
 filename. This data is in a format understod by the pstats module, and
 is generated by a call to the dump_stats() method of profile
 objects. The profile is still shown on screen.

 If you want to run complete programs under the profiler's control, use
 '%run -p [prof_opts] filename.py [args to program]' where prof_opts
 contains profiler specific options as described here.

 You can read the complete documentation for the profile module with:\
   In [1]: import profile; profile.help()
```

%psearch:

```
Search for object in namespaces by wildcard.

%psearch [options] PATTERN [OBJECT TYPE]

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

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

Arguments:
```

```
  PATTERN

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

  [OBJECT TYPE]

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

Options:

  -a: makes the pattern match even objects whose names start with a
  single underscore.  These names are normally ommitted from the
  search.

  -i/-c: make the pattern case insensitive/sensitive.  If neither of
  these options is given, the default is read from your ipythonrc
  file.  The option name which sets this value is
  'wildcards_case_sensitive'.  If this option is not specified in your
  ipythonrc file, IPython's internal default is to do a case sensitive
  search.

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

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

Examples:

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

```
%psearch r*.* string   -> all strings in modules beginning with r

Case sensitve search:

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

Show objects beginning with a single _:

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

%psource:

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

%pushd:

```
Place the current dir on stack and change directory.

Usage:\
  %pushd ['dirname']
```

%pwd:

```
Return the current working directory path.
```

%pycat:

```
Show a syntax-highlighted file through a pager.

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

%quickref:

```
Show a quick reference sheet
```

%quit:

```
Exit IPython, confirming if configured to do so (like %exit)
```

%r:

```
Repeat previous input.

Note: Consider using the more powerfull %rep instead!

If given an argument, repeats the previous command which starts with
the same string, otherwise it just repeats the previous input.
```

```
Shell escaped commands (with ! as first character) are not recognized
by this system, only pure python code and magic commands.
```

%rehashdir:

```
    Add executables in all specified dirs to alias table

Usage:

%rehashdir c:/bin;c:/tools
  - Add all executables under c:/bin and c:/tools to alias table, in
  order to make them directly executable from any directory.

  Without arguments, add all executables in current directory.
```

%rehashx:

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

This version explicitly checks that every entry in $PATH is a file
with execute access (os.X_OK), so it is much slower than %rehash.

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

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

%rep:

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

- %rep (no arguments):

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

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

%rep 45

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

%rep 1-4 6-7 3
```

```
Repeat the specified lines immediately. Input slice syntax is the same as
in %macro and %save.

%rep foo

Place the most recent line that has the substring "foo" to next input.
(e.g. 'svn ci -m foobar').
```

%reset:

```
Resets the namespace by removing all names defined by the user.

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

%run:

```
Run the named file inside IPython as a program.

Usage:\
  %run [-n -i -t [-N<N>] -d [-b<N>] -p [profile options]] file [args]

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

This is similar to running at a system prompt:\
  $ python file args\
but with the advantage of giving you IPython's tracebacks, and of
loading all variables into your interactive namespace for further use
(unless -p is used, see below).

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

Options:

-n: __name__ is NOT set to '__main__', but to the running file's name
without extension (as python does under import).  This allows running
scripts and reloading the definitions in them without calling code
protected by an ' if __name__ == "__main__" ' clause.

-i: run the file in IPython's namespace instead of an empty one. This
is useful if you are experimenting with code written in a text editor
which depends on variables defined interactively.

-e: ignore sys.exit() calls or SystemExit exceptions in the script
```

```
being run.  This is particularly useful if IPython is being used to
run unittests, which always exit with a sys.exit() call.  In such
cases you are interested in the output of the test results, not in
seeing a traceback of the unittest module.

-t: print timing information at the end of the run.  IPython will give
you an estimated CPU time consumption for your script, which under
Unix uses the resource module to avoid the wraparound problems of
time.clock().  Under Unix, an estimate of time spent on system tasks
is also given (for Windows platforms this is reported as 0.0).

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

For example (testing the script uniq_stable.py):

    In [1]: run -t uniq_stable

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

    In [2]: run -t -N5 uniq_stable

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

-d: run your program under the control of pdb, the Python debugger.
This allows you to execute your program step by step, watch variables,
etc.  Internally, what IPython does is similar to calling:

  pdb.run('execfile("YOURFILENAME")')

with a breakpoint set on line 1 of your file.  You can change the line
number for this automatic breakpoint to be <N> by using the -bN option
(where N must be an integer).  For example:

  %run -d -b40 myscript

will set the first breakpoint at line 40 in myscript.py.  Note that
the first breakpoint must be set on a line which actually does
something (not a comment or docstring) for it to stop execution.

When the pdb debugger starts, you will see a (Pdb) prompt.  You must
first enter 'c' (without qoutes) to start execution up to the first
breakpoint.

Entering 'help' gives information about the use of the debugger.  You
can easily see pdb's full documentation with "import pdb;pdb.help()"
```

```
at a prompt.

-p: run program under the control of the Python profiler module (which
prints a detailed report of execution times, function calls, etc).

You can pass other options after -p which affect the behavior of the
profiler itself. See the docs for %prun for details.

In this mode, the program's variables do NOT propagate back to the
IPython interactive namespace (because they remain in the namespace
where the profiler executes them).

Internally this triggers a call to %prun, see its documentation for
details on the options available specifically for profiling.

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

%runlog:

```
Run files as logs.

Usage:\
  %runlog file1 file2 ...

Run the named files (treating them as log files) in sequence inside
the interpreter, and return to the prompt.  This is much slower than
%run because each line is executed in a try/except block, but it
allows running files with syntax errors in them.

Normally IPython will guess when a file is one of its own logfiles, so
you can typically use %run even for logs. This shorthand allows you to
force any file to be treated as a log file.
```

%save:

```
Save a set of lines to a given filename.

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

Options:

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

This function uses the same syntax as %macro for line extraction, but
instead of creating a macro it saves the resulting string to the
filename you specify.
```

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

%sc:

```
Shell capture - execute a shell command and capture its output.

DEPRECATED. Suboptimal, retained for backwards compatibility.

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

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

 "myfiles = !ls ~"

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

--
%sc [options] varname=command

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

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

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

Options:

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

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

In most cases you should not need to split as a list, because the
returned value is a special type of string which can automatically
provide its contents either as a list (split on newlines) or as a
space-separated string.  These are convenient, respectively, either
for sequential processing or to be passed to a shell command.

For example:

    # Capture into variable a
    In [9]: sc a=ls *py

    # a is a string with embedded newlines
    In [10]: a
```

```
    Out[10]: 'setup.py win32_manual_post_install.py'

    # which can be seen as a list:
    In [11]: a.l
    Out[11]: ['setup.py', 'win32_manual_post_install.py']

    # or as a whitespace-separated string:
    In [12]: a.s
    Out[12]: 'setup.py win32_manual_post_install.py'

    # a.s is useful to pass as a single command line:
    In [13]: !wc -l $a.s
      146 setup.py
      130 win32_manual_post_install.py
      276 total

    # while the list form is useful to loop over:
    In [14]: for f in a.l:
       ....:      !wc -l $f
       ....:
    146 setup.py
    130 win32_manual_post_install.py
```

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

```
    In [1]: sc -l b=ls *py

    In [2]: b
    Out[2]: ['setup.py', 'win32_manual_post_install.py']

    In [3]: b.s
    Out[3]: 'setup.py win32_manual_post_install.py'
```

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

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

%store:

```
    Lightweight persistence for python variables.

Example:

ville@badger[~]|1> A = ['hello',10,'world']\
ville@badger[~]|2> %store A\
ville@badger[~]|3> Exit

(IPython session is closed and started again...)
```

```
ville@badger:~$ ipython -p pysh\
ville@badger[~]|1> print A

['hello', 10, 'world']

Usage:

%store          - Show list of all variables and their current values\
%store <var>    - Store the *current* value of the variable to disk\
%store -d <var> - Remove the variable and its value from storage\
%store -z       - Remove all variables from storage\
%store -r       - Refresh all variables from store (delete current vals)\
%store foo >a.txt  - Store value of foo to new file a.txt\
%store foo >>a.txt - Append value of foo to file a.txt\

It should be noted that if you change the value of a variable, you
need to %store it again if you want to persist the new value.

Note also that the variables will need to be pickleable; most basic
python types can be safely %stored.

Also aliases can be %store'd across sessions.
```

%sx:

```
Shell execute - run a shell command and capture its output.

%sx command

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

Notes:

1) If an input line begins with '!!', then %sx is automatically
invoked.  That is, while:
  !ls
causes ipython to simply issue system('ls'), typing
  !!ls
is a shorthand equivalent to:
  %sx ls

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

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

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

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

%system_verbose:

```
Set verbose printing of system calls.

If called without an argument, act as a toggle
```

%time:

```
Time execution of a Python statement or expression.

The CPU and wall clock times are printed, and the value of the
expression (if any) is returned.  Note that under Win32, system time
is always reported as 0, since it can not be measured.

This function provides very basic timing functionality.  In Python
2.3, the timeit module offers more control and sophistication, so this
could be rewritten to use it (patches welcome).

Some examples:

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

  In [2]: n = 1000000

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

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

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

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

```
  Wall time: 0.00 s

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

%timeit:

```
Time execution of a Python statement or expression

Usage:\
  %timeit [-n<N> -r<R> [-t|-c]] statement

Time execution of a Python statement or expression using the timeit
module.

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

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

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

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

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


Examples:\
  In [1]: %timeit pass
  10000000 loops, best of 3: 53.3 ns per loop

  In [2]: u = None

  In [3]: %timeit u is None
  10000000 loops, best of 3: 184 ns per loop

  In [4]: %timeit -r 4 u == None
  1000000 loops, best of 4: 242 ns per loop

  In [5]: import time

  In [6]: %timeit -n1 time.sleep(2)
  1 loops, best of 3: 2 s per loop
```

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

%unalias:

```
Remove an alias
```

%upgrade:

```
 Upgrade your IPython installation

This will copy the config files that don't yet exist in your
ipython dir from the system config dir. Use this after upgrading
IPython if you don't wish to delete your .ipython dir.

Call with -nolegacy to get rid of ipythonrc* files (recommended for
new users)
```

%which:

```
     %which <cmd> => search PATH for files matching cmd. Also scans aliases.

Traverses PATH and prints all files (not just executables!) that match the
pattern on command line. Probably more useful in finding stuff
interactively than 'which', which only prints the first matching item.

Also discovers and expands aliases, so you'll see what will be executed
when you call an alias.

Example:

[~]|62> %which d
d -> ls -F --color=auto
  == c:\cygwin\bin\ls.exe
c:\cygwin\bin\d.exe

[~]|64> %which diff*
diff3 -> diff3
  == c:\cygwin\bin\diff3.exe
diff -> diff
  == c:\cygwin\bin\diff.exe
c:\cygwin\bin\diff.exe
c:\cygwin\bin\diff3.exe
```

%who:

```
Print all interactive variables, with some minimal formatting.

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

  %who function str

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

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

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

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

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

%who_ls:

```
Return a sorted list of all interactive variables.

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

%whos:

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

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

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

  - For {},[],(): their length.

  - For numpy and Numeric arrays, a summary with shape, number of
  elements, typecode and size in memory.

  - Everything else: a string representation, snipping their middle if
  too long.
```

%xmode:

```
Switch modes for the exception handlers.

Valid modes: Plain, Context and Verbose.
```

```
If called without arguments, acts as a toggle.
```

Access to the standard Python help

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

Dynamic object information

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

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

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

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

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

Readline-based features

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

Command line completion

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

Search command history

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

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

Persistent command history across sessions

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

Autoindent

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

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

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

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

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

Customizing readline behavior

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

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

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

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

Session logging and restoring

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

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

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

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

%logstart [log_name [log_mode]]

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

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

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

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

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

System shell access

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

Manual capture of command output

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

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

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

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

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

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

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

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

```
10+y
In [7]: !echo ${x+y}
30
```

Even object attributes can be expanded:

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

System command aliases

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

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

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

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

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

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

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

Recursive reload

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

Verbose and colored exception traceback printouts

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

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

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

Input caching system

IPython offers numbered prompts (In/Out) with input and output caching. All input is saved and can be retrieved as variables (besides the usual arrow key recall).

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

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

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

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

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

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

Output caching system

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

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

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

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

These global variables are all stored in a global dictionary (not a list, since it only has entries for lines which returned a result) available under the names _oh and Out (similar to _ih and In). So the output from line 12

can be obtained as _12, Out[12] or _oh[12]. If you accidentally overwrite the Out variable you can recover it by typing 'Out=_oh' at the prompt.

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

Directory history

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

Automatic parentheses and quotes

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

Automatic parentheses

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

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

and the input will be translated to this:

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

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

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

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

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

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

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

but this will work:

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

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

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

Automatic quoting

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

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

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

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

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

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

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

3.2.3 IPython as your default Python environment

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

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

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

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

3.2.4 Embedding IPython

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

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

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

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

```
from IPython.Shell import IPShellEmbed

ipshell = IPShellEmbed()

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

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

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

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

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

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

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

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

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

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

# The basics to get you going:
```

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

# Try running this code both at the command line and from inside IPython (with
# %run example-embed.py)
try:
    __IPYTHON__
except NameError:
    nested = 0
    args = ['']
else:
    print "Running nested copies of IPython."
    print "The prompts for the nested copy have been modified"
    nested = 1
    # what the embedded instance will see as sys.argv:
    args = ['-pi1','In <\\#>: ','-pi2','   .\\D.: ',
            '-po','Out<\\#>: ','-nosep']

# First import the embeddable shell class
from IPython.Shell import IPShellEmbed

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

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

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

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

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

#---------------------------------------------------------------------------
# More details:

# IPShellEmbed instances don't print the standard system banner and
# messages. The IPython banner (which actually may contain initialization
```

```
# messages) is available as <instance>.IP.BANNER in case you want it.

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

# - A global startup banner.

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

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

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

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

# The code below illustrates all this.


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

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

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

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

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

# Reactivate the IPython shell
```

```
ipshell.set_dummy_mode(0)

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

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

print 'Main program finished. Bye!'

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

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

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

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

#-----------------------------------------------------------------------------
# This code loads IPython but modifies a few things if it detects it's running
# embedded in another IPython session (helps avoid confusion)

try:
    __IPYTHON__
except NameError:
    argv = ['']
    banner = exit_msg = ''
else:
    # Command-line options for IPython (a list like sys.argv)
    argv = ['-pi1','In <\\#>:','-pi2','   .\\D.:','-po','Out<\\#>:']
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'

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

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

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

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

```
# dummy function.

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

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

3.2.5 Using the Python debugger (pdb)

Running entire programs via pdb

pdb, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of pdb, regardless of whether you have wrapped it into a 'main()' function or not. For this, simply type '%run -d myscript' at an IPython prompt. See the %run command's documentation (via '%run?' or in Sec. magic for more details, including how to control where pdb will stop execution first.

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

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

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

Automatic invocation of pdb on exceptions

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

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

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

```
import sys,IPython.ultraTB
sys.excepthook = IPython.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.

3.2.6 Extensions for syntax processing

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

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

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

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

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

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

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

```
IPython profile: tutorial

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

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

```
   ...: ...      return result
   ...:

In [2]: fib2(10)
Out[2]: [1, 1, 2, 3, 5, 8]
```

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

Input of physical quantities with units

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

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

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

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

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

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

3.2.7 Threading support

WARNING: The threading support is still somewhat experimental, and it has only seen reasonable testing under Linux. Threaded code is particularly tricky to debug, and it tends to show extremely platform-dependent behavior. Since I only have access to Linux machines, I will have to rely on user's experiences and assistance for this area of IPython to improve under other platforms.

IPython, via the -gthread , -qthread, -q4thread and -wthread options (described in Sec. Threading options), can run in multithreaded mode to support pyGTK, Qt3, Qt4 and WXPython applications respectively. These GUI toolkits need to control the python main loop of execution, so under a normal Python interpreter, starting a pyGTK, Qt3, Qt4 or WXPython application will immediately freeze the shell.

IPython, with one of these options (you can only use one at a time), separates the graphical loop and IPython's code execution run into different threads. This allows you to test interactively (with %run, for example) your GUI code without blocking.

A nice mini-tutorial on using IPython along with the Qt Designer application is available at the SciPy wiki: http://www.scipy.org/Cookbook/Matplotlib/Qt_with_IPython_and_Designer.

Tk issues

As indicated in Sec. Threading options, a special -tk option is provided to try and allow Tk graphical applications to coexist interactively with WX, Qt or GTK ones. Whether this works at all, however, is very platform and configuration dependent. Please experiment with simple test cases before committing to using this combination of Tk and GTK/Qt/WX threading in a production environment.

I/O pitfalls

Be mindful that the Python interpreter switches between threads every N bytecodes, where the default value as of Python 2.3 is $N=100.$ This value can be read by using the sys.getcheckinterval() function, and it can be reset via sys.setcheckinterval(N). This switching of threads can cause subtly confusing effects if one of your threads is doing file I/O. In text mode, most systems only flush file buffers when they encounter a 'n'. An instruction as simple as:

```
print >> filehandle, ''hello world''
```

actually consists of several bytecodes, so it is possible that the newline does not reach your file before the next thread switch. Similarly, if you are writing to a file in binary mode, the file won't be flushed until the buffer fills, and your other thread may see apparently truncated files.

For this reason, if you are using IPython's thread support and have (for example) a GUI application which will read data generated by files written to from the IPython thread, the safest approach is to open all of your files in unbuffered mode (the third argument to the file/open function is the buffering value):

```
filehandle = open(filename,mode,0)
```

This is obviously a brute force way of avoiding race conditions with the file buffering. If you want to do it cleanly, and you have a resource which is being shared by the interactive IPython loop and your GUI thread, you should really handle it with thread locking and syncrhonization properties. The Python documentation discusses these.

3.2.8 Interactive demos with IPython

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

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

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

Any python script can be run as a demo, but that does little more than showing
it on-screen, syntax-highlighted in one shot.  If you add a little simple
```

```
markup, you can stop at specified intervals and return to the ipython prompt,
resuming execution later.
"""

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

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

x = 1
y = 2

# <demo> --- stop ---

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

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

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

print 'z=',x

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

print 'bye!'
```

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

```
from IPython.demo import Demo

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

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

```
mydemo
```

and IPython will call it, executing each block. Demo objects can be restarted, you can move forward or back skipping blocks, re-execute the last block, etc. Simply use the Tab key on a demo object to see its

methods, and call '?' on them to see their docstrings for more usage details. In addition, the demo module itself contains a comprehensive docstring, which you can access via:

```
from IPython import demo

demo?
```

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

3.2.9 Plotting with matplotlib

The matplotlib library (http://matplotlib.sourceforge.net http://matplotlib.sourceforge.net) provides high quality 2D plotting for Python. Matplotlib can produce plots on screen using a variety of GUI toolkits, including Tk, GTK 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.

IPython accepts the special option -pylab (see *here*). This configures it to support matplotlib, honoring the settings in the .matplotlibrc file. IPython will detect the user's choice of matplotlib GUI backend, and automatically select the proper threading model to prevent blocking. It also sets matplotlib in interactive mode and modifies %run slightly, so that any matplotlib-based script can be executed using %run and the final show() command does not block the interactive shell.

The -pylab option must be given first in order for IPython to configure its threading mode. However, you can still issue other options afterwards. This allows you to have a matplotlib-based environment customized with additional modules using the standard IPython profile mechanism (see *here*): `ipython -pylab -p myprofile` will load the profile defined in ipythonrc-myprofile after configuring matplotlib.

3.3 IPython as a system shell

3.3.1 Overview

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

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

The 'sh' profile is different from the default profile in that:

- Prompt shows the current directory

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

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

- '$$var = command' and '$var = command' syntax is not supported
- anymore. Use 'var = !command' instead (incidentally, this is
- available in all IPython profiles). Note that !!command *will*
- work.

3.3.2 Aliases

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

3.3.3 Directory management

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

IPython provides its own builtin '%cd' magic command to move in the filesystem (the % is not required with 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.

3.3.4 Enabled extensions

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

envpersist

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

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

ipy_which

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

Example:

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

ipy_app_completers

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

ipy_rehashdir

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

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

Now, you can execute any of those commams directly:

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

(the putty window opens).

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

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

mglob

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

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

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

3.3.5 Prompt customization

The sh profile uses the following prompt configurations:

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

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

3.3.6 String lists

String lists (IPython.genutils.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 setedito
[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')
```

3.3.7 Real world example: remove all files outside version control

First, capture output of "hg status":

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

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

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

Now we can just remove these files by doing 'rm $junk.s'.

3.3.8 The .s, .n, .p properties

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

3.4 IPython extension API

IPython api (defined in IPython/ipapi.py) is the public api that should be used for

- Configuration of user preferences (.ipython/ipy_user_conf.py)
- Creating new profiles (.ipython/ipy_profile_PROFILENAME.py)
- Writing extensions

Note that by using the extension api for configuration (editing ipy_user_conf.py instead of ipythonrc), you get better validity checks and get richer functionality - for example, you can import an extension and call functions in it to configure it for your purposes.

For an example extension (the 'sh' profile), see IPython/Extensions/ipy_profile_sh.py.

For the last word on what's available, see the source code of IPython/ipapi.py.

3.4.1 Getting started

If you want to define an extension, create a normal python module that can be imported. The module will access IPython functionality through the 'ip' object defined below.

If you are creating a new profile (e.g. foobar), name the module as 'ipy_profile_foobar.py' and put it in your ~/.ipython directory. Then, when you start ipython with the '-p foobar' argument, the module is automatically imported on ipython startup.

If you are just doing some per-user configuration, you can either

- Put the commands directly into ipy_user_conf.py.
- Create a new module with your customization code and import *that* module in ipy_user_conf.py. This is preferable to the first approach, because now you can reuse and distribute your customization code.

3.4.2 Getting a handle to the api

Put this in the start of your module:

```
#!python
import IPython.ipapi
ip = IPython.ipapi.get()
```

The 'ip' object will then be used for accessing IPython functionality. 'ip' will mean this api object in all the following code snippets. The same 'ip' that we just acquired is always accessible in interactive IPython sessions by the name _ip - play with it like this:

```
[~\_ipython]|81> a = 10
[~\_ipython]|82> _ip.e
_ip.ev           _ip.ex           _ip.expose_magic
[~\_ipython]|82> _ip.ev('a+13')
            <82> 23
```

The _ip object is also used in some examples in this document - it can be substituted by 'ip' in non-interactive use.

3.4.3 Changing options

The ip object has 'options' attribute that can be used te get/set configuration options (just as in the ipythonrc file):

```
o = ip.options
o.autocall = 2
o.automagic = 1
```

3.4.4 Executing statements in IPython namespace with 'ex' and 'ev'

Often, you want to e.g. import some module or define something that should be visible in IPython namespace. Use `ip.ev` to *evaluate* (calculate the value of) expression and `ip.ex` to ''execute'' a statement:

```
# path module will be visible to the interactive session
ip.ex("from path import path" )

# define a handy function 'up' that changes the working directory

ip.ex('import os')
ip.ex("def up(): os.chdir('..')")


# _i2 has the input history entry #2, print its value in uppercase.
print ip.ev('_i2.upper()')
```

3.4.5 Accessing the IPython namespace

ip.user_ns attribute has a dictionary containing the IPython global namespace (the namespace visible in the interactive session).

```
[~\_ipython]|84> tauno = 555
[~\_ipython]|85> _ip.user_ns['tauno']
            <85> 555
```

3.4.6 Defining new magic commands

The following example defines a new magic command, %impall. What the command does should be obvious:

```
def doimp(self, arg):
    ip = self.api
    ip.ex("import %s; reload(%s); from %s import *" % (
```

```
    arg,arg,arg)
    )

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

Things to observe in this example:

- Define a function that implements the magic command using the ipapi methods defined in this document
- The first argument of the function is 'self', i.e. the interpreter object. It shouldn't be used directly. however. The interpreter object is probably *not* going to remain stable through IPython versions.
- Access the ipapi through 'self.api' instead of the global 'ip' object.
- All the text following the magic command on the command line is contained in the second argument
- Expose the magic by ip.expose_magic()

3.4.7 Calling magic functions and system commands

Use ip.magic() to execute a magic function, and ip.system() to execute a system command:

```
# go to a bookmark
ip.magic('%cd -b relfiles')

# execute 'ls -F' system command. Interchangeable with os.system('ls'), really.
ip.system('ls -F')
```

3.4.8 Launching IPython instance from normal python code

Use ipapi.launch_new_instance() with an argument that specifies the namespace to use. This can be useful for trivially embedding IPython into your program. Here's an example of normal python program test.py (''without'' an existing IPython session) that launches an IPython interpreter and regains control when the interpreter is exited:

```
[ipython]|1> cat test.py
my_ns = dict(
    kissa = 15,
    koira = 16)
import IPython.ipapi
print "launching IPython instance"
IPython.ipapi.launch_new_instance(my_ns)
print "Exited IPython instance!"
print "New vals:",my_ns['kissa'], my_ns['koira']
```

And here's what it looks like when run (note how we don't start it from an ipython session):

```
Q:\ipython>python test.py
launching IPython instance
Py 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] IPy 0.7.3b3.r1975
[ipython]|1> kissa = 444
[ipython]|2> koira = 555
[ipython]|3> Exit
Exited IPython instance!
New vals: 444 555
```

3.4.9 Accessing unexposed functionality

There are still many features that are not exposed via the ipapi. If you can't avoid using them, you can use the functionality in InteractiveShell object (central IPython session class, defined in iplib.py) through ip.IP.

For example:

```
[~]|7> _ip.IP.expand_aliases('np','myfile.py')
   <7> 'c:/opt/Notepad++/notepad++.exe myfile.py'
[~]|8>
```

Still, it's preferable that if you encounter such a feature, contact the IPython team and request that the functionality be exposed in a future version of IPython. Things not in ipapi are more likely to change over time.

3.4.10 Provided extensions

You can see the list of available extensions (and profiles) by doing `import ipy_<TAB>`. Some extensions don't have the `ipy_` prefix in module name, so you may need to see the contents of IPython/Extensions folder to see what's available.

You can see a brief documentation of an extension by looking at the module docstring:

```
[c:p/ipython_main]|190> import ipy_fsops
[c:p/ipython_main]|191> ipy_fsops?

...

Docstring:
    File system operations

Contains: Simple variants of normal unix shell commands (icp, imv, irm,
imkdir, igrep).
```

You can also install your own extensions - the recommended way is to just copy the module to ~/.ipython. Extensions are typically enabled by just importing them (e.g. in ipy_user_conf.py), but some extensions require additional steps, for example:

```
[c:p]|192> import ipy_traits_completer
[c:p]|193> ipy_traits_completer.activate()
```

Note that extensions, even if provided in the stock IPython installation, are not guaranteed to have the same requirements as the rest of IPython - an extension may require external libraries or a newer version of Python than what IPython officially requires. An extension may also be under a more restrictive license than IPython (e.g. ipy_bzr is under GPL).

Just for reference, the list of bundled extensions at the time of writing is below:

astyle.py clearcmd.py envpersist.py ext_rescapture.py ibrowse.py igrid.py InterpreterExec.py InterpreterPasteInput.py ipipe.py ipy_app_completers.py ipy_autoreload.py ipy_bzr.py ipy_completers.py ipy_constants.py ipy_defaults.py ipy_editors.py ipy_exportdb.py ipy_extutil.py ipy_fsops.py ipy_gnuglobal.py ipy_kitcfg.py ipy_legacy.py ipy_leo.py ipy_p4.py ipy_profile_doctest.py ipy_profile_none.py ipy_profile_scipy.py ipy_profile_sh.py ipy_profile_zope.py ipy_pydb.py ipy_rehashdir.py ipy_render.py ipy_server.py ipy_signals.py ipy_stock_completers.py ipy_system_conf.py ipy_traits_completer.py ipy_vimserver.py ipy_which.py ipy_workdir.py jobctrl.py ledit.py numeric_formats.py PhysicalQInput.py PhysicalQInteractive.py pickleshare.py pspersistence.py win32clip.py __init__.py

CHAPTER

FOUR

Using IPython for parallel computing

4.1 Overview and getting started

Contents

4.1.1 Introduction

This file 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.

4.1.2 Architecture overview

The IPython architecture consists of three components:

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

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

IPython engine

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

IPython controller

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

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

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

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

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

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

Controller clients

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

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

Security

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

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

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

Currently, there are three .furl files that the controller creates:

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

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

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

More details of how these `.furl` files are used are given below.

4.1.3 Getting Started

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

Starting the controller and engine on your local machine

This is the simplest configuration that can be used and is useful for testing the system and on machines that have multiple cores and/or multple CPUs. The easiest way of getting started is to use the **ipcluster** command:

```
$ ipcluster -n 4
```

This will start an IPython controller and then 4 engines that connect to the controller. Lastly, the script will print out the Python commands that you can use to connect to the controller. It is that easy.

> **Warning:** The **ipcluster** does not currently work on Windows. We are working on it though.

Underneath the hood, the controller creates `.furl` files in the `~./ipython/security` directory. Because the engines are on the same host, they automatically find the needed `ipcontroller-engine.furl` there and use it to connect to the controller.

The **ipcluster** script uses two other top-level scripts that you can also use yourself. These scripts are **ipcontroller**, which starts the controller and **ipengine** which starts one engine. To use these scripts to start things on your local machine, do the following.

First start the controller:

```
$ ipcontroller
```

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

```
$ ipengine
```

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

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

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

Starting the controller and engines on different hosts

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

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

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

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

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

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

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

Make .furl files persistent

At fist glance it may seem that that managing the `.furl` files is a bit annoying. Going back to the house and key analogy, copying the `.furl` around each time you start the controller is like having to make a new

key everytime you want to unlock the door and enter your house. As with your house, you want to be able to create the key (or `.furl` file) once, and then simply use it at any point in the future.

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

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

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

Note: You may ask the question: what ports does the controller listen on if you don't tell 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 obcurity and ii) to multiple controllers on a given host to start and automatically use different ports.

Starting engines using `mpirun`

The IPython engines can be started using `mpirun`/`mpiexec`, even if the engines don't call `MPI_Init()` or use the MPI API in any way. This is supported on modern MPI implementations like Open MPI.. This provides an really nice way of starting a bunch of engine. On a system with MPI installed you can do:

```
mpirun -n 4 ipengine
```

to start 4 engine on a cluster. This works even if you don't have any Python-MPI bindings installed.

More details on using MPI with IPython can be found *here*.

Log files

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

4.1.4 Next Steps

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

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

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

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

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

```
[0] Out[1]: Hello World

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

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

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

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

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

Same thing hold true of creating a task client:

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

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

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

4.2 IPython's multiengine interface

Contents

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

4.2.1 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 -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

4.2.2 Creating a `MultiEngineClient` instance

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

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

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

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

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

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

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

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

4.2.3 Quick and easy parallelism

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

Parallel map

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

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

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

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

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

See Also:

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

Parallel function decorator

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

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

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

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

4.2.4 Running Python commands

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

Blocking execution

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

```
# The default is to run on all engines
In [4]: mec.execute('a=5')
Out[4]:
<Results List>
[0] In [1]: a=5
[1] In [1]: a=5
[2] In [1]: a=5
[3] In [1]: a=5

In [5]: mec.execute('b=10')
Out[5]:
<Results List>
[0] In [2]: b=10
[1] In [2]: b=10
[2] In [2]: b=10
[3] In [2]: b=10
```

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

```
In [6]: mec.execute('c=a+b',targets=[0,2])
Out[6]:
<Results List>
[0] In [3]: c=a+b
[2] In [3]: c=a+b


In [7]: mec.execute('c=a-b',targets=[1,3])
Out[7]:
<Results List>
[1] In [3]: c=a-b
[3] In [3]: c=a-b


In [8]: mec.execute('print c')
Out[8]:
<Results List>
[0] In [4]: print c
[0] Out[4]: 15

[1] In [4]: print c
[1] Out[4]: -5

[2] In [4]: print c
[2] Out[4]: 15

[3] In [4]: print c
[3] Out[4]: -5
```

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

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

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

Non-blocking execution

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

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

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

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

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

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

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

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

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

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

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

# Then, their results are ready using get_result or the r attribute
In [75]: pr_list[0].r
Out[75]:
<Results List>
```

```
[0] In [20]: time.sleep(3)
[1] In [19]: time.sleep(3)
[2] In [20]: time.sleep(3)
[3] In [19]: time.sleep(3)
```

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

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

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

The following examples demonstrate how to use the instance attributes:

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

In [17]: mec.block = False

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

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

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

In [21]: mec.block = True

In [22]: mec.execute('b=10; print b')
Out[22]:
<Results List>
[0] In [7]: b=10; print b
[0] Out[7]: 10

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

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

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

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

Parallel magic commands

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

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

In [24]: mec.block=True

In [25]: import numpy

In [26]: %px import numpy
Executing command on Controller
Out[26]:
<Results List>
[0] In [8]: import numpy
[1] In [7]: import numpy
[2] In [8]: import numpy
[3] In [7]: import numpy


In [27]: %px a = numpy.random.rand(2,2)
Executing command on Controller
Out[27]:
<Results List>
[0] In [9]: a = numpy.random.rand(2,2)
[1] In [8]: a = numpy.random.rand(2,2)
[2] In [9]: a = numpy.random.rand(2,2)
[3] In [8]: a = numpy.random.rand(2,2)


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

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

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

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

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

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

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

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

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

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

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

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

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

In [33]: for i in range(100):
   ....:     a = numpy.random.rand(10,10)
   ....:     a = a+a.transpose()
   ....:     evals = numpy.linalg.eigvals(a)
   ....:     max_evals.append(evals[0].real)
   ....:
   ....:
<IPython.kernel.multiengineclient.PendingResult object at 0x17af8f0>

In [34]: %autopx
Auto Parallel Disabled

In [35]: mec.block=True

In [36]: px print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
Executing command on Controller
Out[36]:
<Results List>
[0] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[0] Out[13]: Average max eigenvalue is:  10.1387247332

[1] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[1] Out[12]: Average max eigenvalue is:  10.2076902286
```

```
[2] In [13]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[2] Out[13]: Average max eigenvalue is:  10.1891484655

[3] In [12]: print "Average max eigenvalue is: ", sum(max_evals)/len(max_evals)
[3] Out[12]: Average max eigenvalue is:  10.1158837784
```

4.2.5 Moving Python objects around

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

Basic push and pull

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

```
In [38]: mec.push(dict(a=1.03234,b=3453))
Out[38]: [None, None, None, None]

In [39]: mec.pull('a')
Out[39]: [1.03234, 1.03234, 1.03234, 1.03234]

In [40]: mec.pull('b',targets=0)
Out[40]: [3453]

In [41]: mec.pull(('a','b'))
Out[41]: [[1.03234, 3453], [1.03234, 3453], [1.03234, 3453], [1.03234, 3453]]

In [42]: mec.zip_pull(('a','b'))
Out[42]: [(1.03234, 1.03234, 1.03234, 1.03234), (3453, 3453, 3453, 3453)]

In [43]: mec.push(dict(c='speed'))
Out[43]: [None, None, None, None]

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

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

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

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

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

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

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

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

Push and pull for functions

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

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

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

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

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


    In [56]: px print y
    Executing command on Controller
    Out[56]:
    <Results List>
    [0] In [16]: print y
    [0] Out[16]: 512.0

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

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

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

Dictionary interface

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

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

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

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

Scatter and gather

Sometimes it is useful to partition a sequence and push the partitions to different engines. In MPI language, this is know as scatter/gather and we follow that terminology. However, it is important to remember that in IPython's `MultiEngineClient` class, `scatter()` is from the interactive IPython session to the engines and `gather()` is from the engines back to the interactive IPython session. For scatter/gather operations between engines, MPI should be used:

```
In [58]: mec.scatter('a',range(16))
Out[58]: [None, None, None, None]

In [59]: px print a
Executing command on Controller
Out[59]:
<Results List>
[0] In [17]: print a
[0] Out[17]: [0, 1, 2, 3]

[1] In [16]: print a
[1] Out[16]: [4, 5, 6, 7]

[2] In [17]: print a
[2] Out[17]: [8, 9, 10, 11]

[3] In [16]: print a
[3] Out[16]: [12, 13, 14, 15]


In [60]: mec.gather('a')
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

4.2.6 Other things to look at

How to do parallel list comprehensions

In many cases list comprehensions are nicer than using the map function. While we don't have fully parallel list comprehensions, it is simple to get the basic effect using `scatter()` and `gather()`:

```
In [66]: mec.scatter('x',range(64))
Out[66]: [None, None, None, None]

In [67]: px y = [i**10 for i in x]
Executing command on Controller
Out[67]:
<Results List>
[0] In [19]: y = [i**10 for i in x]
[1] In [18]: y = [i**10 for i in x]
[2] In [19]: y = [i**10 for i in x]
[3] In [18]: y = [i**10 for i in x]


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

In [69]: print y
[0, 1, 1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824,...]
```

Parallel exceptions

In the multiengine interface, parallel commands can raise Python exceptions, just like serial commands. But, it is a little subtle, because a single parallel command can actually raise multiple exceptions (one for each engine the command was run on). To express this idea, the MultiEngine interface has a `CompositeError` exception class that will be raised in most cases. The `CompositeError` class is a special type of exception that wraps one or more other types of exceptions. Here is how it works:

```
In [76]: mec.block=True

In [77]: mec.execute('1/0')
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in execute(self, lines, target
    432         targets, block = self._findTargetsAndBlock(targets, block)
    433         result = blockingCallFromThread(self.smultiengine.execute, lines,
--> 434             targets=targets, block=block)
    435         if block:
    436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
     72             result.raiseException()
```

```
     73         except Exception, e:
---> 74             raise e
     75     return result
     76

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

Notice how the error message printed when `CompositeError` is raised has information about the individual exceptions that were raised on each engine. If you want, you can even raise one of these original exceptions:

```
In [80]: try:
   ....:     mec.execute('1/0')
   ....: except client.CompositeError, e:
   ....:     e.raise_exception()
   ....:
   ....:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/error.pyc in raise_exception(self, excid)
    156             raise IndexError("an exception with index %i does not exist"%excid)
    157         else:
--> 158             raise et, ev, etb
    159
    160 def collect_exceptions(rlist, method):

ZeroDivisionError: integer division or modulo by zero
```

If you are working in IPython, you can simple type `%debug` after one of these `CompositeError` exceptions is raised, and inspect the exception instance:

```
In [81]: mec.execute('1/0')
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in execute(self, lines, target
    432         targets, block = self._findTargetsAndBlock(targets, block)
    433         result = blockingCallFromThread(self.smultiengine.execute, lines,
--> 434             targets=targets, block=block)
    435         if block:
    436             result = ResultList(result)

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
```

```
     72             result.raiseException()
     73         except Exception, e:
---> 74             raise e
     75     return result
     76

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

In [82]: %debug
>

/ipython1-client-r3021/ipython1/kernel/twistedutil.py(74)blockingCallFromThread()
     73         except Exception, e:
---> 74             raise e
     75     return result

# With the debugger running, e is the exceptions instance.  We can tab complete
# on it and see the extra methods that are available.
ipdb> e.
e.__class__          e.__getitem__       e.__new__           e.__setstate__      e.args
e.__delattr__        e.__getslice__      e.__reduce__        e.__str__           e.elist
e.__dict__           e.__hash__          e.__reduce_ex__     e.__weakref__       e.message
e.__doc__            e.__init__          e.__repr__          e._get_engine_str   e.print_tra
e.__getattribute__   e.__module__        e.__setattr__       e._get_traceback    e.raise_exc
ipdb> e.print_tracebacks()
[0:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero

[1:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero

[2:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero
```

```
[3:execute]:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<string> in <module>()

ZeroDivisionError: integer division or modulo by zero
```

Note: The above example appears to be broken right now because of a change in how we are using Twisted.

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

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

In [84]: pr = mec.execute('1/0')

In [85]: pr.r
---------------------------------------------------------------------------
CompositeError                            Traceback (most recent call last)

/ipython1-client-r3021/docs/examples/<ipython console> in <module>()

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in _get_r(self)
    170
    171     def _get_r(self):
--> 172         return self.get_result(block=True)
    173
    174     r = property(_get_r)

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_result(self, default, k
    131                 return self.result
    132         try:
--> 133             result = self.client.get_pending_deferred(self.result_id, block)
    134         except error.ResultNotCompleted:
    135             return default

/ipython1-client-r3021/ipython1/kernel/multiengineclient.pyc in get_pending_deferred(self,
    385
    386     def get_pending_deferred(self, deferredID, block):
--> 387         return blockingCallFromThread(self.smultiengine.get_pending_deferred, defe
    388
    389     def barrier(self, pendingResults):

/ipython1-client-r3021/ipython1/kernel/twistedutil.pyc in blockingCallFromThread(f, *a, **k
     72             result.raiseException()
     73         except Exception, e:
---> 74             raise e
     75     return result
     76

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

```
[2:execute]: ZeroDivisionError: integer division or modulo by zero
[3:execute]: ZeroDivisionError: integer division or modulo by zero
```

4.3 The IPython task interface

Contents

The task interface to the controller presents the engines as a fault tolerant, dynamic load-balanced system or workers. Unlike the multiengine interface, in the task interface, the user have no direct access to individual engines. In some ways, this interface is simpler, but in other ways it is more powerful.

Best of all the user can use both of these interfaces running at the same time to take advantage or both of their strengths. When the user can break up the user's work into segments that do not depend on previous execution, the task interface is ideal. But it also has more power and flexibility, allowing the user to guide the distribution of jobs, without having to assign tasks to engines explicitly.

4.3.1 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 -n 4
```

For more detailed information about starting the controller and engines, see our *introduction* to using IPython for parallel computing.

4.3.2 Creating a `TaskClient` instance

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

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

In [2]: tc = client.TaskClient()
```

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

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

4.3.3 Quick and easy parallelism

In many cases, you simply want to apply a Python function to a sequence of objects, but *in parallel*. Like the multiengine interface, the task interface provides two simple ways of accomplishing this: a parallel version of `map()` and `@parallel` function decorator. However, the verions in the task interface have one important difference: they are dynamically load balanced. Thus, if the execution time per item varies significantly, you should use the versions in the task interface.

Parallel map

The parallel `map()` in the task interface is similar to that in the multiengine interface:

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

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

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

Parallel function decorator

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

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

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

4.3.4 More details

The `TaskClient` has many more powerful features that allow quite a bit of flexibility in how tasks are defined and run. The next places to look are in the following classes:

- `IPython.kernel.client.TaskClient`
- `IPython.kernel.client.StringTask`
- `IPython.kernel.client.MapTask`

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

1. Create a `TaskClient`.
2. Create one or more instances of `StringTask` or `MapTask` to define your tasks.
3. Submit your tasks to using the `run()` method of your `TaskClient` instance.
4. Use `TaskClient.get_task_result()` to get the results of the tasks.

We are in the process of developing more detailed information about the task interface. For now, the docstrings of the `TaskClient`, `StringTask` and `MapTask` classes should be consulted.

4.4 Using MPI with IPython

The simplest way of getting started with MPI is to install an MPI implementation (we recommend Open MPI) and mpi4py and then start the engines using the `mpirun` command:

```
mpirun -n 4 ipengine --mpi=mpi4py
```

This will automatically import mpi4py and make sure that *MPI_Init* is called at the right time. We also have built in support for PyTrilinos, which can be used (assuming PyTrilinos is installed) by starting the engines with:

```
mpirun -n 4 ipengine --mpi=pytrilinos
```

CHAPTER

FIVE

Configuration and customization

5.1 Initial configuration of your environment

This section will help you set various things in your environment for your IPython sessions to be as efficient as possible. All of IPython's configuration information, along with several example files, is stored in a directory named by default $HOME/.ipython. You can change this by defining the environment variable IPYTHONDIR, or at runtime with the command line option -ipythondir.

If all goes well, the first time you run IPython it should automatically create a user copy of the config directory for you, based on its builtin defaults. You can look at the files it creates to learn more about configuring the system. The main file you will modify to configure IPython's behavior is called ipythonrc (with a .ini extension under Windows), included for reference *here*. This file is very commented and has many variables you can change to suit your taste, you can find more details *here*. Here we discuss the basic things you will want to make sure things are working properly from the beginning.

5.1.1 Access to the Python help system

This is true for Python in general (not just for IPython): you should have an environment variable called PYTHONDOCS pointing to the directory where your HTML Python documentation lives. In my system it's `/usr/share/doc/python-doc/html`, check your local details or ask your systems administrator.

This is the directory which holds the HTML version of the Python manuals. Unfortunately it seems that different Linux distributions package these files differently, so you may have to look around a bit. Below I show the contents of this directory on my system for reference:

```
[html]> ls
about.html  dist/  icons/      lib/           python2.5.devhelp.gz  whatsnew/
acks.html   doc/   index.html  mac/           ref/
api/        ext/   inst/       modindex.html  tut/
```

You should really make sure this variable is correctly set so that Python's pydoc-based help system works. It is a powerful and convenient system with full access to the Python manuals and all modules accessible to you.

Under Windows it seems that pydoc finds the documentation automatically, so no extra setup appears necessary.

5.1.2 Editor

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.

If you are a dedicated Emacs user, 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))
```

You can also set the value of this editor via the commmand-line option '-editor' or in your ipythonrc file. This is useful if you wish to use specifically for IPython an editor different from your typical default (and for Windows users who tend to use fewer environment variables).

5.1.3 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.

5.1.4 Input/Output prompts and exception tracebacks

You can test whether the colored prompts and tracebacks work on your system interactively by typing '%colors Linux' at the prompt (use '%colors LightBG' if your terminal has a light background). If the input prompt shows garbage like:

```
[0;32mIn [[1;32m1[0;32m]: [0;00m
```

instead of (in color) something like:

```
In [1]:
```

this means that your terminal doesn't properly handle color escape sequences. You can go to a 'no color' mode by typing '%colors NoColor'.

You can try using a different terminal emulator program (Emacs users, see below). To permanently set your color preferences, edit the file $HOME/.ipython/ipythonrc and set the colors option to the desired value.

5.1.5 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 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.

5.1.6 (X)Emacs configuration

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

Important 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 documentation directory (where this manual resides in PDF and HTML formats).

Once you put these files in your Emacs path, all you need in your .emacs file is:

```
(require 'ipython)
```

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

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

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

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

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

Notes:

- 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 customize py-python-command previously, this value will override what ipython.el does (because loading the customization variables comes later).

5.2 Customization of IPython

There are 2 ways to configure IPython - the old way of using ipythonrc files (an INI-file like format), and the new way that involves editing your ipy_user_conf.py. Both configuration systems work at the same time, so you can set your options in both, but if you are hesitating about which alternative to choose, we recommend the ipy_user_conf.py approach, as it will give you more power and control in the long run. However, there are few options such as pylab_import_all that can only be specified in ipythonrc file or command line - the reason for this is that they are needed before IPython has been started up, and the IPApi object used in ipy_user_conf.py is not yet available at that time. A hybrid approach of specifying a few options in ipythonrc and doing the more advanced configuration in ipy_user_conf.py is also possible.

5.2.1 The ipythonrc approach

As we've already mentioned, IPython reads a configuration file which can be specified at the command line (-rcfile) or which by default is assumed to be called ipythonrc. Such a file is looked for in the current directory where IPython is started and then in your IPYTHONDIR, which allows you to have local configuration files for specific projects. In this section we will call these types of configuration files simply rcfiles (short for resource configuration file).

The syntax of an rcfile is one of key-value pairs separated by whitespace, one per line. Lines beginning with a # are ignored as comments, but comments can not be put on lines with data (the parser is fairly primitive). Note that these are not python files, and this is deliberate, because it allows us to do some things which would be quite tricky to implement if they were normal python files.

First, an rcfile can contain permanent default values for almost all command line options (except things like -help or -Version). *This section* contains a description of all command-line options. However, values you explicitly specify at the command line override the values defined in the rcfile.

Besides command line option values, the rcfile can specify values for certain extra special options which are not available at the command line. These options are briefly described below.

Each of these options may appear as many times as you need it in the file.

- include <file1> <file2> ...: you can name other rcfiles you want to recursively load up to 15 levels (don't use the <> brackets in your names!). This feature allows you to define a 'base' rcfile with general options and special-purpose files which can be loaded only when needed with particular configuration options. To make this more convenient, IPython accepts the -profile <name> option (abbreviates to -p <name>) which tells it to look for an rcfile named ipythonrc-<name>.
- import_mod <mod1> <mod2> ...: import modules with 'import <mod1>,<mod2>,...'
- import_some <mod> <f1> <f2> ...: import functions with 'from <mod> import <f1>,<f2>,...'

- import_all <mod1> <mod2> ...: for each module listed import functions with `from <mod> import *`.
- execute <python code>: give any single-line python code to be executed.
- execfile <filename>: execute the python file given with an 'execfile(filename)' command. Username expansion is performed on the given names. So if you need any amount of extra fancy customization that won't fit in any of the above 'canned' options, you can just put it in a separate python file and execute it.
- alias <alias_def>: this is equivalent to calling '%alias <alias_def>' at the IPython command line. This way, from within IPython you can do common system tasks without having to exit it or use the ! escape. IPython isn't meant to be a shell replacement, but it is often very useful to be able to do things with files while testing code. This gives you the flexibility to have within IPython any aliases you may be used to under your normal system shell.

5.2.2 ipy_user_conf.py

There should be a simple template ipy_user_conf.py file in your ~/.ipython directory. It is a plain python module that is imported during IPython startup, so you can do pretty much what you want there - import modules, configure extensions, change options, define magic commands, put variables and functions in the IPython namespace, etc. You use the IPython extension api object, acquired by IPython.ipapi.get() and documented in the "IPython extension API" chapter, to interact with IPython. A sample ipy_user_conf.py is listed below for reference:

```
# Most of your config files and extensions will probably start
# with this import

import IPython.ipapi
ip = IPython.ipapi.get()

# You probably want to uncomment this if you did %upgrade -nolegacy
# import ipy_defaults

import os

def main():

    #ip.dbg.debugmode = True
    ip.dbg.debug_stack()

    # uncomment if you want to get ipython -p sh behaviour
    # without having to use command line switches
    import ipy_profile_sh
    import jobctrl

    # Configure your favourite editor?
    # Good idea e.g. for %edit os.path.isfile

    #import ipy_editors
```

```
    # Choose one of these:

    #ipy_editors.scite()
    #ipy_editors.scite('c:/opt/scite/scite.exe')
    #ipy_editors.komodo()
    #ipy_editors.idle()
    # ... or many others, try 'ipy_editors??' after import to see them

    # Or roll your own:
    #ipy_editors.install_editor("c:/opt/jed +$line $file")


    o = ip.options
    # An example on how to set options
    #o.autocall = 1
    o.system_verbose = 0

    #import_all("os sys")
    #execf('~/_ipython/ns.py')


    # -- prompt
    # A different, more compact set of prompts from the default ones, that
    # always show your current location in the filesystem:

    #o.prompt_in1 = r'\C_LightBlue[\C_LightCyan\Y2\C_LightBlue]\C_Normal\n\C_Green|\#>'
    #o.prompt_in2 = r'.\D: '
    #o.prompt_out = r'[\#] '

    # Try one of these color settings if you can't read the text easily
    # autoexec is a list of IPython commands to execute on startup
    #o.autoexec.append('%colors LightBG')
    #o.autoexec.append('%colors NoColor')
    o.autoexec.append('%colors Linux')


# some config helper functions you can use
def import_all(modules):
    """ Usage: import_all("os sys") """
    for m in modules.split():
        ip.ex("from %s import *" % m)

def execf(fname):
    """ Execute a file in user namespace """
    ip.ex('execfile("%s")' % os.path.expanduser(fname))

main()
```

5.2.3 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
```

5.2.4 IPython profiles

As we already mentioned, IPython supports the -profile command-line option (see *here*). A profile is nothing more than a particular configuration file like your basic ipythonrc one, but with particular customizations for a specific purpose. When you start IPython with 'ipython -profile <name>', it assumes that in your IPYTHONDIR there is a file called ipythonrc-<name> or ipy_profile_<name>.py, and loads it instead of the normal ipythonrc.

This system allows you to maintain multiple configurations which load modules, set options, define functions, etc. suitable for different tasks and activate them in a very simple manner. In order to avoid having to repeat all of your basic options (common things that don't change such as your color preferences, for example), any profile can include another configuration file. The most common way to use profiles is then to have each one include your basic ipythonrc file as a starting point, and then add further customizations.

5.3 New configuration system

IPython has a configuration system. When running IPython for the first time, reasonable defaults are used for the configuration. The configuration of IPython can be changed in two ways:

- Configuration files
- Commands line options (which override the configuration files)

IPython has a separate configuration file for each subpackage. Thus, the main configuration files are (in your `~/.ipython` directory):

- `ipython1.core.ini`
- `ipython1.kernel.ini`
- `ipython1.notebook.ini`

To create these files for the first time, do the following:

```
from IPython.kernel.config import config_manager as kernel_config
kernel_config.write_default_config_file()
```

But, you should only need to do this if you need to modify the defaults. If needed repeat this process with the `notebook` and `core` configuration as well. If you are running into problems with IPython, you might try deleting these configuration files.

CHAPTER

SIX

What's new

Contents

6.1 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.

6.2 Release 0.9

6.2.1 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 mulitple variable follows that of *map*, iii) both the multiengine and task controller now have a parallel function implementation.
- All of the parallel computing capabilities from *ipython1-dev* have been merged into IPython proper. This resulted in the following new subpackages: `IPython.kernel`, `IPython.kernel.core`, `IPython.config`, `IPython.tools` and `IPython.testing`.
- As part of merging in the *ipython1-dev* stuff, the *setup.py* script and friends have been completely refactored. Now we are checking for dependencies using the approach that matplotlib uses.

- The documentation has been completely reorganized to accept the documentation from *ipython1-dev*.
- We have switched to using Foolscap for all of our network protocols in `IPython.kernel`. This gives us secure connections that are both encrypted and authenticated.
- We have a brand new *COPYING.txt* files that describes the IPython license and copyright. The biggest change is that we are putting "The IPython Development Team" as the copyright holder. We give more details about exactly what this means in this file. All developer should read this and use the new banner in all IPython source code files.
- sh profile: ./foo runs foo as system command, no need to do !./foo anymore
- String lists now support `sort(field, nums = True)` method (to easily sort system command output). Try it with `a = !ls -l ; a.sort(1, nums=1)`.
- '%cpaste foo' now assigns the pasted block as string list, instead of string
- The ipcluster script now run by default with no security. This is done because the main usage of the script is for starting things on localhost. Eventually when ipcluster is able to start things on other hosts, we will put security back.
- 'cd –foo' searches directory history for string foo, and jumps to that dir. Last part of dir name is checked first. If no matches for that are found, look at the whole path.

6.2.2 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.

6.2.3 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.

6.2.4 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`.

6.3 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.

6.4 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.

6.5 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.

6.6 Older releases

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

CHAPTER

SEVEN

Development

7.1 IPython development guidelines

7.1.1 Overview

IPython is the next generation of IPython. It is named such for two reasons:

- Eventually, IPython will become IPython version 1.0.
- This new code base needs to be able to co-exist with the existing IPython until it is a full replacement for it. Thus we needed a different name. We couldn't use `ipython` (lowercase) as some files systems are case insensitive.

There are two, no three, main goals of the IPython effort:

1. Clean up the existing codebase and write lots of tests.
2. Separate the core functionality of IPython from the terminal to enable IPython to be used from within a variety of GUI applications.
3. Implement a system for interactive parallel computing.

While the third goal may seem a bit unrelated to the main focus of IPython, it turns out that the technologies required for this goal are nearly identical with those required for goal two. This is the main reason the interactive parallel computing capabilities are being put into IPython proper. Currently the third of these goals is furthest along.

This document describes IPython from the perspective of developers.

7.1.2 Project organization

Subpackages

IPython is organized into semi self-contained subpackages. Each of the subpackages will have its own:

- **Dependencies**. One of the most important things to keep in mind in partitioning code amongst subpackages, is that they should be used to cleanly encapsulate dependencies.

- **Tests**. Each subpackage shoud have its own `tests` subdirectory that contains all of the tests for that package. For information about writing tests for IPython, see the Testing System section of this document.
- **Configuration**. Each subpackage should have its own `config` subdirectory that contains the configuration information for the components of the subpackage. For information about how the IPython configuration system works, see the Configuration System section of this document.
- **Scripts**. Each subpackage should have its own `scripts` subdirectory that contains all of the command line scripts associated with the subpackage.

Installation and dependencies

IPython will not use setuptools for installation. Instead, we will use standard `setup.py` scripts that use distutils. While there are a number a extremely nice features that setuptools has (like namespace packages), the current implementation of setuptools has performance problems, particularly on shared file systems. In particular, when Python packages are installed on NSF file systems, import times become much too long (up towards 10 seconds).

Because IPython is being used extensively in the context of high performance computing, where performance is critical but shared file systems are common, we feel these performance hits are not acceptable. Thus, until the performance problems associated with setuptools are addressed, we will stick with plain distutils. We are hopeful that these problems will be addressed and that we will eventually begin using setuptools. Because of this, we are trying to organize IPython in a way that will make the eventual transition to setuptools as painless as possible.

Because we will be using distutils, there will be no method for automatically installing dependencies. Instead, we are following the approach of Matplotlib which can be summarized as follows:

- Distinguish between required and optional dependencies. However, the required dependencies for IPython should be only the Python standard library.
- Upon installation check to see which optional dependencies are present and tell the user which parts of IPython need which optional dependencies.

It is absolutely critical that each subpackage of IPython has a clearly specified set of dependencies and that dependencies are not carelessly inherited from other IPython subpackages. Furthermore, tests that have certain dependencies should not fail if those dependencies are not present. Instead they should be skipped and print a message.

Specific subpackages

`core` This is the core functionality of IPython that is independent of the terminal, network and GUIs. Most of the code that is in the current IPython trunk will be refactored, cleaned up and moved here.

`kernel` The enables the IPython core to be expose to a the network. This is also where all of the parallel computing capabilities are to be found.

`config` The configuration package used by IPython.

frontends The various frontends for IPython. A frontend is the end-user application that exposes the capabilities of IPython to the user. The most basic frontend will simply be a terminal based application that looks just like today 's IPython. Other frontends will likely be more powerful and based on GUI toolkits.

notebook An application that allows users to work with IPython notebooks.

tools This is where general utilities go.

7.1.3 Version control

In the past, IPython development has been done using Subversion. Recently, we made the transition to using Bazaar and Launchpad. This makes it much easier for people to contribute code to IPython. Here is a sketch of how to use Bazaar for IPython development. First, you should install Bazaar. After you have done that, make sure that it is working by getting the latest main branch of IPython:

```
$ bzr branch lp:ipython
```

Now you can create a new branch for you to do your work in:

```
$ bzr branch ipython ipython-mybranch
```

The typical work cycle in this branch will be to make changes in `ipython-mybranch` and then commit those changes using the commit command:

```
$ ...do work in ipython-mybranch...
$ bzr ci -m "the commit message goes here"
```

Please note that since we now don't use an old-style linear ChangeLog (that tends to cause problems with distributed version control systems), you should ensure that your log messages are reasonably detailed. Use a docstring-like approach in the commit messages (including the second line being left *blank*):

```
Single line summary of  changes being committed.

- more details when warranted ...
- including crediting outside contributors if they sent the
  code/bug/idea!
```

If we couple this with a policy of making single commits for each reasonably atomic change, the bzr log should give an excellent view of the project, and the *–short* log option becomes a nice summary.

While working with this branch, it is a good idea to merge in changes that have been made upstream in the parent branch. This can be done by doing:

```
$ bzr pull
```

If this command shows that the branches have diverged, then you should do a merge instead:

```
$ bzr merge lp:ipython
```

If you want others to be able to see your branch, you can create an account with launchpad and push the branch to your own workspace:

```
$ bzr push bzr+ssh://<me>@bazaar.launchpad.net/~<me>/+junk/ipython-mybranch
```

Finally, once the work in your branch is done, you can merge your changes back into the *ipython* branch by using merge:

```
$ cd ipython
$ merge ../ipython-mybranch
[resolve any conflicts]
$ bzr ci -m "Fixing that bug"
$ bzr push
```

But this will require you to have write permissions to the *ipython* branch. It you don't you can tell one of the IPython devs about your branch and they can do the merge for you.

More information about Bazaar workflows can be found here.

7.1.4 Documentation

Standalone documentation

All standalone documentation should be written in plain text (`.txt`) files using reStructuredText for markup and formatting. All such documentation should be placed in the top level directory `docs` of the IPython source tree. Or, when appropriate, a suitably named subdirectory should be used. The documentation in this location will serve as the main source for IPython documentation and all existing documentation should be converted to this format.

In the future, the text files in the `docs` directory will be used to generate all forms of documentation for IPython. This include documentation on the IPython website as well as *pdf* documentation.

Docstring format

Good docstrings are very important. All new code will use Epydoc for generating API docs, so we will follow the Epydoc conventions. More specifically, we will use reStructuredText for markup and formatting, since it is understood by a wide variety of tools. This means that if in the future we have any reason to change from Epydoc to something else, we'll have fewer transition pains.

Details about using reStructuredText for docstrings can be found here.

Additional PEPs of interest regarding documentation of code:

- Docstring Conventions
- Docstring Processing System Framework
- Docutils Design Specification

7.1.5 Coding conventions

General

In general, we'll try to follow the standard Python style conventions as described here:

- Style Guide for Python Code

Other comments:

- In a large file, top level classes and functions should be separated by 2-3 lines to make it easier to separate them visually.
- Use 4 spaces for indentation.
- Keep the ordering of methods the same in classes that have the same methods. This is particularly true for classes that implement similar interfaces and for interfaces that are similar.

Naming conventions

In terms of naming conventions, we'll follow the guidelines from the Style Guide for Python Code.

For all new IPython code (and much existing code is being refactored), we'll use:

- All `lowercase` module names.
- `CamelCase` for class names.
- `lowercase_with_underscores` for methods, functions, variables and attributes.

This may be confusing as most of the existing IPython codebase uses a different convention (`lowerCamelCase` for methods and attributes). Slowly, we will move IPython over to the new convention, providing shadow names for backward compatibility in public interfaces.

There are, however, some important exceptions to these rules. In some cases, IPython code will interface with packages (Twisted, Wx, Qt) that use other conventions. At some level this makes it impossible to adhere to our own standards at all times. In particular, when subclassing classes that use other naming conventions, you must follow their naming conventions. To deal with cases like this, we propose the following policy:

- If you are subclassing a class that uses different conventions, use its naming conventions throughout your subclass. Thus, if you are creating a Twisted Protocol class, used Twisted's `namingSchemeForMethodsAndAttributes.`
- All IPython's official interfaces should use our conventions. In some cases this will mean that you need to provide shadow names (first implement `fooBar` and then `foo_bar = fooBar`). We want to avoid this at all costs, but it will probably be necessary at times. But, please use this sparingly!

Implementation-specific *private* methods will use `_single_underscore_prefix`. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).

Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like `runlines()` where `runsource()` and `runcode()` had established precedent).

The old IPython codebase has a big mix of classes and modules prefixed with an explicit `IP`. In Python this is mostly unnecessary, redundant and frowned upon, as namespaces offer cleaner prefixing. The only case where this approach is justified is for classes which are expected to be imported into external namespaces and a very generic name (like Shell) is too likely to clash with something else. We'll need to revisit this issue as we clean up and refactor the code, but in general we should remove as many unnecessary `IP`/`ip` prefixes as possible. However, if a prefix seems absolutely necessary the more specific `IPY` or `ipy` are preferred.

7.1.6 Testing system

It is extremely important that all code contributed to IPython has tests. Tests should be written as unittests, doctests or as entities that the Nose testing package will find. Regardless of how the tests are written, we will use Nose for discovering and running the tests. Nose will be required to run the IPython test suite, but will not be required to simply use IPython.

Tests of Twisted using code should be written by subclassing the `TestCase` class that comes with `twisted.trial.unittest`. When this is done, Nose will be able to run the tests and the twisted reactor will be handled correctly.

Each subpackage in IPython should have its own `tests` directory that contains all of the tests for that subpackage. This allows each subpackage to be self-contained. If a subpackage has any dependencies beyond the Python standard library, the tests for that subpackage should be skipped if the dependencies are not found. This is very important so users don't get tests failing simply because they don't have dependencies.

We also need to look into use Noses ability to tag tests to allow a more modular approach of running tests.

7.1.7 Configuration system

IPython uses .ini files for configuration purposes. This represents a huge improvement over the configuration system used in IPython. IPython works with these files using the ConfigObj package, which IPython includes as `ipython1/external/configobj.py`.

Currently, we are using raw ConfigObj objects themselves. Each subpackage of IPython should contain a `config` subdirectory that contains all of the configuration information for the subpackage. To see how configuration information is defined (along with defaults) see at the examples in `ipython1/kernel/config` and `ipython1/core/config`. Likewise, to see how the configuration information is used, see examples in `ipython1/kernel/scripts/ipengine.py`.

Eventually, we will add a new layer on top of the raw ConfigObj objects. We are calling this new layer, `tconfig`, as it will use a Traits-like validation model. We won't actually use Traits, but will implement something similar in pure Python. But, even in this new system, we will still use ConfigObj and .ini files underneath the hood. Talk to Fernando if you are interested in working on this part of IPython. The current prototype of `tconfig` is located in the IPython sandbox.

7.1.8 Installation and testing scenarios

This section outlines the various scenarios that we need to test before we release an IPython version. These scenarios represent different ways of installing IPython and its dependencies.

Installation scenarios under Linux and OS X

1. Install from tarball using `python setup.py install`. a. With only readline+nose dependencies installed. b. With all dependencies installed (readline, zope.interface, Twisted, foolscap, Sphinx, nose, pyOpenSSL).
2. Install using easy_install.

   a. With only readline+nose dependencies installed. i. Default dependencies: `easy_install ipython-0.9.beta3-py2.5.egg` ii. Optional dependency sets: `easy_install -f ipython-0.9.beta3-py2.5.egg IPython[kernel,doc,test,security]`

   b. With all dependencies already installed.

Installation scenarios under Win32

1. Install everything from .exe installers
2. easy_install?

Tests to run for these scenarios

1. Run the full test suite.
2. Start a controller and engines and try a few things by hand. a. Using ipcluster. b. Using ipcontroller/ipengine by hand.
3. Run a few of the parallel examples.
4. Try the kernel with and without security with and without PyOpenSSL installed.
5. Beat on the IPython terminal a bunch.
6. Make sure that furl files are being put in proper locations.

7.1.9 Release checklist

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

1. Run the release script, which makes the tar.gz, eggs and Win32 .exe installer. It posts them to the site and registers the release with PyPI.
2. Updating the website with announcements and links to the updated changes.txt in html form. Remember to put a short note both on the news page of the site and on launcphad.

3. Drafting a short release announcement with i) highlights and ii) a link to the html changes.txt.
4. Make sure that the released version of the docs is live on the site.
5. Celebrate!

7.2 Development roadmap

Contents

IPython is an ambitious project that is still under heavy development. However, we want IPython to become useful to as many people as possible, as quickly as possible. To help us accomplish this, we are laying out a roadmap of where we are headed and what needs to happen to get there. Hopefully, this will help the IPython developers figure out the best things to work on for each upcoming release.

Speaking of releases, we are going to begin releasing a new version of IPython every four weeks. We are hoping that a regular release schedule, along with a clear roadmap of where we are headed will propel the project forward.

7.2.1 Where are we headed

Our goal with IPython is simple: to provide a *powerful*, *robust* and *easy to use* framework for parallel computing. While there are other secondary goals you will hear us talking about at various times, this is the primary goal of IPython that frames the roadmap.

7.2.2 Steps along the way

Here we describe the various things that we need to work on to accomplish this goal.

Setting up for regular release schedule

We would like to begin to release IPython regularly (probably a 4 week release cycle). To get ready for this, we need to revisit the development guidelines and put in information about releasing IPython.

Process startup and management

IPython is implemented using a distributed set of processes that communicate using TCP/IP network channels. Currently, users have to start each of the various processes separately using command line scripts. This is both difficult and error prone. Furthermore, there are a number of things that often need to be managed once the processes have been started, such as the sending of signals and the shutting down and cleaning up of processes.

We need to build a system that makes it trivial for users to start and manage IPython processes. This system should have the following properties:

- It should possible to do everything through an extremely simple API that users can call from their own Python script. No shell commands should be needed.
- This simple API should be configured using standard .ini files.
- The system should make it possible to start processes using a number of different approaches: SSH, PBS/Torque, Xgrid, Windows Server, mpirun, etc.
- The controller and engine processes should each have a daemon for monitoring, signaling and clean up.
- The system should be secure.
- The system should work under all the major operating systems, including Windows.

Initial work has begun on the daemon infrastructure, and some of the needed logic is contained in the ipcluster script.

Ease of use/high-level approaches to parallelism

While our current API for clients is well designed, we can still do a lot better in designing a user-facing API that is super simple. The main goal here is that it should take *almost no extra code* for users to get their code running in parallel. For this to be possible, we need to tie into Python's standard idioms that enable efficient coding. The biggest ones we are looking at are using context managers (i.e., Python 2.5's `with` statement) and decorators. Initial work on this front has begun, but more work is needed.

We also need to think about new models for expressing parallelism. This is fun work as most of the foundation has already been established.

Security

Currently, IPython has no built in security or security model. Because we would like IPython to be usable on public computer systems and over wide area networks, we need to come up with a robust solution for security. Here are some of the specific things that need to be included:

- User authentication between all processes (engines, controller and clients).
- Optional TSL/SSL based encryption of all communication channels.

- A good way of picking network ports so multiple users on the same system can run their own controller and engines without interfering with those of others.
- A clear model for security that enables users to evaluate the security risks associated with using IPython in various manners.

For the implementation of this, we plan on using Twisted's support for SSL and authentication. One things that we really should look at is the Foolscap network protocol, which provides many of these things out of the box.

The security work needs to be done in conjunction with other network protocol stuff.

As of the 0.9 release of IPython, we are using Foolscap and we have implemented a full security model.

Latent performance issues

Currently, we have a number of performance issues that are waiting to bite users:

- The controller store a large amount of state in Python dictionaries. Under heavy usage, these dicts with get very large, causing memory usage problems. We need to develop more scalable solutions to this problem, such as using a sqlite database to store this state. This will also help the controller to be more fault tolerant.
- Currently, the client to controller connections are done through XML-RPC using HTTP 1.0. This is very inefficient as XML-RPC is a very verbose protocol and each request must be handled with a new connection. We need to move these network connections over to PB or Foolscap. Done!
- We currently don't have a good way of handling large objects in the controller. The biggest problem is that because we don't have any way of streaming objects, we get lots of temporary copies in the low-level buffers. We need to implement a better serialization approach and true streaming support.
- The controller currently unpickles and repickles objects. We need to use the [push|pull]_serialized methods instead.
- Currently the controller is a bottleneck. We need the ability to scale the controller by aggregating multiple controllers into one effective controller.

7.3 IPython.kernel.core.notification blueprint

7.3.1 Overview

The `IPython.kernel.core.notification` module will provide a simple implementation of a notification center and support for the observer pattern within the `IPython.kernel.core`. The main intended use case is to provide notification of Interpreter events to an observing frontend during the execution of a single block of code.

7.3.2 Functional Requirements

The notification center must:

- Provide synchronous notification of events to all registered observers.

- Provide typed or labeled notification types
- Allow observers to register callbacks for individual or all notification types
- Allow observers to register callbacks for events from individual or all notifying objects
- Notification to the observer consists of the notification type, notifying object and user-supplied extra information [implementation: as keyword parameters to the registered callback]
- Perform as O(1) in the case of no registered observers.
- Permit out-of-process or cross-network extension.

7.3.3 What's not included

As written, the `IPython.kernel.core.notificaiton` module does not:

- Provide out-of-process or network notifications [these should be handled by a separate, Twisted aware module in `IPython.kernel`].
- Provide zope.interface-style interfaces for the notification system [these should also be provided by the `IPython.kernel` module]

7.3.4 Use Cases

The following use cases describe the main intended uses of the notificaiton module and illustrate the main success scenario for each use case:

1. Dwight Schroot is writing a frontend for the IPython project. His frontend is stuck in the stone age and must communicate synchronously with an IPython.kernel.core.Interpreter instance. Because code is executed in blocks by the Interpreter, Dwight's UI freezes every time he executes a long block of code. To keep track of the progress of his long running block, Dwight adds the following code to his frontend's set-up code:

   ```
   from IPython.kernel.core.notification import NotificationCenter
   center = NotificationCenter.sharedNotificationCenter
   center.registerObserver(self, type=IPython.kernel.core.Interpreter.STDOUT_NOTIFICA
   ```

   and elsewhere in his front end:

   ```
   def stdout_notification(self, type, notifying_object, out_string=None):
       self.writeStdOut(out_string)
   ```

   If everything works, the Interpreter will (according to its published API) fire a notification via the `IPython.kernel.core.notification.sharedCenter` of type `STD_OUT_NOTIFICATION_TYPE` before writing anything to stdout [it's up to the Intereter implementation to figure out when to do this]. The notificaiton center will then call the registered callbacks for that event type (in this case, Dwight's frontend's stdout_notification method). Again, according to its API, the Interpreter provides an additional keyword argument when firing the notificaiton of out_string, a copy of the string it will write to stdout.

   Like magic, Dwight's frontend is able to provide output, even during long-running calculations. Now if Jim could just convince Dwight to use Twisted...

1. Boss Hog is writing a frontend for the IPython project. Because Boss Hog is stuck in the stone age, his frontend will be written in a new Fortran-like dialect of python and will run only from the command line. Because he doesn't need any fancy notification system and is used to worrying about every cycle on his rat-wheel powered mini, Boss Hog is adamant that the new notification system not produce any performance penalty. As they say in Hazard county, there's no such thing as a free lunch. If he wanted zero overhead, he should have kept using IPython 0.8. Instead, those tricky Duke boys slide in a suped-up bridge-out jumpin' awkwardly confederate-lovin' notification module that imparts only a constant (and small) performance penalty when the Interpreter (or any other object) fires an event for which there are no registered observers. Of course, the same notificaiton-enabled Interpreter can then be used in frontends that require notifications, thus saving the IPython project from a nasty civil war.
2. Barry is wrting a frontend for the IPython project. Because Barry's front end is the *new hotness*, it uses an asynchronous event model to communicate with a Twisted `engineservice` that communicates with the IPython `Interpreter`. Using the `IPython.kernel.notification` module, an asynchronous wrapper on the `IPython.kernel.core.notification` module, Barry's frontend can register for notifications from the interpreter that are delivered asynchronously. Even if Barry's frontend is running on a separate process or even host from the Interpreter, the notifications are delivered, as if by dark and twisted magic. Just like Dwight's frontend, Barry's frontend can now recieve notifications of e.g. writing to stdout/stderr, opening/closing an external file, an exception in the executing code, etc.

CHAPTER

EIGHT

Frequently asked questions

8.1 General questions

8.2 Questions about parallel computing with IPython

8.2.1 Will IPython speed my Python code up?

Yes and no. When converting a serial code to run in parallel, there often many difficulty questions that need to be answered, such as:

- How should data be decomposed onto the set of processors?
- What are the data movement patterns?
- Can the algorithm be structured to minimize data movement?
- Is dynamic load balancing important?

We can't answer such questions for you. This is the hard (but fun) work of parallel computing. But, once you understand these things IPython will make it easier for you to implement a good solution quickly. Most importantly, you will be able to use the resulting parallel code interactively.

With that said, if your problem is trivial to parallelize, IPython has a number of different interfaces that will enable you to parallelize things is almost no time at all. A good place to start is the `map` method of our `MultiEngineClient`.

8.2.2 What is the best way to use MPI from Python?

8.2.3 What about all the other parallel computing packages in Python?

Some of the unique characteristic of IPython are:

- IPython is the only architecture that abstracts out the notion of a parallel computation in such a way that new models of parallel computing can be explored quickly and easily. If you don't like the models we provide, you can simply create your own using the capabilities we provide.
- IPython is asynchronous from the ground up (we use Twisted).

- IPython's architecture is designed to avoid subtle problems that emerge because of Python's global interpreter lock (GIL).
- While IPython's architecture is designed to support a wide range of novel parallel computing models, it is fully interoperable with traditional MPI applications.
- IPython has been used and tested extensively on modern supercomputers.
- IPython's networking layers are completely modular. Thus, is straightforward to replace our existing network protocols with high performance alternatives (ones based upon Myranet/Infiniband).
- IPython is designed from the ground up to support collaborative parallel computing. This enables multiple users to actively develop and run the *same* parallel computation.
- Interactivity is a central goal for us. While IPython does not have to be used interactivly, it can be.

8.2.4 Why The IPython controller a bottleneck in my parallel calculation?

A golden rule in parallel computing is that you should only move data around if you absolutely need to. The main reason that the controller becomes a bottleneck is that too much data is being pushed and pulled to and from the engines. If your algorithm is structured in this way, you really should think about alternative ways of handling the data movement. Here are some ideas:

1. Have the engines write data to files on the locals disks of the engines.
2. Have the engines write data to files on a file system that is shared by the engines.
3. Have the engines write data to a database that is shared by the engines.
4. Simply keep data in the persistent memory of the engines and move the computation to the data (rather than the data to the computation).
5. See if you can pass data directly between engines using MPI.

8.2.5 Isn't Python slow to be used for high-performance parallel computing?

CHAPTER

NINE

History

9.1 Origins

IPython was starting in 2001 by Fernando Perez. IPython as we know it today grew out of the following three projects:

- ipython by Fernando Pérez. I was working on adding Mathematica-type prompts and a flexible configuration system (something better than $PYTHONSTARTUP) to the standard Python interactive interpreter.
- IPP by Janko Hauser. Very well organized, great usability. Had an old help system. IPP was used as the 'container' code into which I 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 it:

> 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.

9.2 Today and how we got here

This needs to be filled in.

CHAPTER

TEN

License and Copyright

10.1 License

IPython is licensed under the terms of the new or revised BSD license, as follows:

```
Copyright (c) 2008, IPython Development Team

All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of
conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list
of conditions and the following disclaimer in the documentation and/or other
materials provided with the distribution.

Neither the name of the IPython Development Team nor the names of its contributors
may be used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

10.2 About the IPython Development Team

Fernando Perez began IPython in 2001 based on code from Janko Hauser <jhauser@zscout.de> and Nathaniel Gray <n8gray@caltech.edu>. Fernando is still the project lead.

The IPython Development Team is the set of all contributors to the IPython project. This includes all of the IPython subprojects. Here is a list of the currently active contributors:

- Matthieu Brucher
- Ondrej Certik
- Laurent Dufrechou
- Robert Kern
- Brian E. Granger
- Fernando Perez (project leader)
- Benjamin Ragan-Kelley
- Ville M. Vainio
- Gael Varoquux
- Stefan van der Walt
- Tech-X Corporation
- Barry Wark

If your name is missing, please add it.

10.3 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 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.

10.4 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.

CHAPTER

ELEVEN

Credits

IPython is led by Fernando Pérez.

As of this writing, the following developers have joined the core team:

- [Robert Kern] <rkern-AT-enthought.com>: co-mentored the 2005 Google Summer of Code project to develop python interactive notebooks (XML documents) and graphical interface. This project was awarded to the students Tzanko Matev <tsanko-AT-gmail.com> and Toni Alatalo <antont-AT-an.org>.
- [Brian Granger] <ellisonbg-AT-gmail.com>: extending IPython to allow support for interactive parallel computing.
- [Benjamin (Min) Ragan-Kelley]: key work on IPython's parallel computing infrastructure.
- [Ville Vainio] <vivainio-AT-gmail.com>: Ville has made many improvements to the core of IPython and was the maintainer of the main IPython trunk from version 0.7.1 to 0.8.4.
- [Gael Varoquaux] <gael.varoquaux-AT-normalesup.org>: work on the merged architecture for the interpreter as of version 0.9, implementing a new WX GUI based on this system.
- [Barry Wark] <barrywark-AT-gmail.com>: implementing a new Cocoa GUI, as well as work on the new interpreter architecture and Twisted support.
- [Laurent Dufrechou] <laurent.dufrechou-AT-gmail.com>: development of the WX GUI support.
- [Jörgen Stenarson] <jorgen.stenarson-AT-bostream.nu>: maintainer of the PyReadline project, necessary for IPython under windows.

The IPython project is also very grateful to:

Bill Bumgarner <bbum-AT-friday.com>: for providing the DPyGetOpt module which gives very powerful and convenient handling of command-line options (light years ahead of what Python 2.1.1's getopt module does).

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 (bugs are still my fault, not his).

Obviously Guido van Rossum and the whole Python development team, that goes without saying.

IPython's website is generously hosted at http://ipython.scipy.orgby Enthought (http://www.enthought.com). I am very grateful to them and all of the SciPy team for their contribution.

Fernando would also like to thank Stephen Figgins <fig-AT-monitor.net>, an O'Reilly Python editor. His Oct/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.

And last but not least, all the kind IPython users who have emailed new code, bug reports, fixes, comments and ideas. A brief list follows, please let us know if we have ommitted your name by accident:

- Dan Milstein <danmil-AT-comcast.net>. A bold refactoring of the core prefilter stuff 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 IPyhton, including making necessary changes to matplotlib itself.
- [Matthew Arnison] <maffew-AT-cat.org.au> Bug reports, '%run -d' idea.
- [Prabhu Ramachandran] <prabhu_r-AT-users.sourceforge.net> Help with (X)Emacs support, threading patches, ideas...
- [Norbert Tretkowski] <tretkowski-AT-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>: support for the new config system.

INDEX